Join the Learn Enough Society today to get access to all tutorials and screencasts (including the Rails 5 edition of the Rails Tutorial)

Learn Enough CSS & Layout to Be Dangerous

Lee Donahoe and Michael Hartl

An introduction to CSS and page layout, including static site generation with Jekyll

---

About the authors

Learn Enough cofounder Lee Donahoe is an entrepreneur, designer, and front-end developer. In addition to doing the design for Learn Enough, Softcover, and the Ruby on Rails Tutorial, he is also a cofounder and front-end developer for Coveralls, a leading test coverage analysis service, and is tech cofounder of Buck Mason, a men's clothing company once featured on ABC's *Shark Tank*. Lee is a graduate of USC, where he majored in Economics and studied Interactive Multimedia & Technologies.

Michael Hartl is the creator of the *Ruby on Rails Tutorial*, one of the leading introductions to web development, and is cofounder and principal author at Learn Enough to Be Dangerous. Previously, he was a physics instructor at the California Institute of Technology (Caltech), where he received a Lifetime Achievement Award for Excellence in Teaching. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Chapter 1
Introduction to CSS

Welcome to *Learn Enough CSS & Layout to Be Dangerous*!

CSS—short for *Cascading Style Sheets*—is the styling language of the World Wide Web. CSS lets developers and designers define what a web page looks like and how it behaves, including how elements are positioned in the browser. Every website that you visit (with some incredibly rare exceptions) uses CSS to make the user experience and interface look inviting, which means that learning the basics of CSS is an essential part of becoming a web developer or designer. It's also

useful for anyone who *interacts* with developers and designers, which these days seems like practically everyone.

Most CSS tutorials teach the subject in isolation, showing you how to make individual changes to things like text color or font size, without showing you how to put everything together as an integrated whole. This approach is a lot like trying to teach a foreign language by having students read the dictionary. You might learn lots of words, but you would end up with little to no context or ability to have a real conversation (Figure 1.1).[1]

In contrast, *Learn Enough CSS & Layout to Be Dangerous* is specifically designed to show you how CSS works in the context of a real website.

Figure 1.1: You're not going to learn Spanish from the dictionary alone.

It's especially hard to find tutorials that cover how to develop the *layout* of the page—that is, how one positions elements on the page and determines what content goes where. It's because this subject is so important—and yet so neglected—that we called this tutorial *Learn Enough CSS & Layout to Be Dangerous*, rather than simply *Learn Enough CSS to Be Dangerous*.

Why is layout so often ignored? It's partially because the layout aspects of CSS can be rather complicated, but it's also because doing layout right calls for more than plain HTML and CSS. Making a real, industrial-grade website requires using a *templating system* to assemble the various parts—there are simply too many repeated and custom elements (such as headers, footers, dynamically generated names and dates, etc.) to build such sites by hand (Figure 1.2).

Figure 1.2: Most modern websites have many repeated and custom elements.

As a result, even learning enough CSS to be *dangerous*—that is, enough to accomplish your goals without taking the time to become a complete expert—is far more extensive than most people realize. You need to learn not only the basic CSS rules, but also more advanced rules governing page layout, together with a tool to assemble all the parts into a combined whole.

Learn Enough CSS & Layout to Be Dangerous is designed to fill this need. It fits into the Learn Enough introductory sequence immediately after *Learn Enough HTML to Be Dangerous*, but it's also suitable for more experienced developers who want to strengthen their knowledge of web design.[2] Among other things, this means that *Learn Enough CSS & Layout to Be Dangerous* is both a possible prerequisite and also an excellent follow-on to the *Ruby on Rails Tutorial*.

As a result of building on the ideas developed in previous Learn Enough tutorials (Box 1.1), *Learn Enough CSS & Layout to Be Dangerous* is also unusual in that we will be deploying our sample CSS site to the live Web, all while following professional-grade development practices. Because of this integrated approach, *Learn Enough CSS & Layout to Be Dangerous* puts everything together in a way you probably haven't seen before, even if you've previously studied CSS.

Box 1.1. Prerequisites.

This tutorial assumes that you know how to use a Unix command line, are comfortable using a text editor, are familiar with using Git for version control, and understand the basic structure of an HTML page. If you'd like an introduction or a refresher on any of these subjects, we suggest reading one or more of the corresponding Learn Enough tutorials (all of which are available for free online, and are available with videos via a subscription to the Learn Enough Society):

1. *Learn Enough Command Line to Be Dangerous*
2. *Learn Enough Text Editor to Be Dangerous*
3. *Learn Enough Git to Be Dangerous*
4. *Learn Enough HTML to Be Dangerous*

You'll also probably find it helpful to set up your computer as a proper development environment, as covered in *Learn Enough Dev Environment to Be Dangerous*, although this step can be deferred until Chapter 5.

1.1 You're a front-end developer

CSS isn't the kind of language that's useful to learn in little half-steps—the part that people find difficult is dealing with handling styling when you are deep into creating an honest-to-goodness website with many levels of complexity. The real skill comes in knowing how to plan for a multi-page site that uses a bunch of different snippets of code[3] placed into a flexible layout that organizes content and data in a useful way.

There are already many sites (like the Mozilla Developer Network CSS Reference) that exhaustively document every CSS property, so going over each and every option from the specification would just be a duplication of effort. Instead, *Learn Enough CSS & Layout to Be Dangerous* is designed to complement such reference works by showing you how CSS applies to the design of a real website. The resulting narrative explanation gives you the context necessary to understand and apply CSS documentation—especially when combined with a little technical sophistication (Box 1.2).

Box 1.2. Technical sophistication

One of the principal themes of the Learn Enough tutorials is the development of *technical sophistication*, the ability to figure out and solve technical problems.

This tutorial includes many opportunities to apply technical sophistication. For example, many of the code listings require you to orient yourself in a CSS file and figure out where to put the new style rules introduced by the listing. Another important technique is learning to *comment out* CSS rules and then refreshing the browser to see what they do. We'll also occasionally add CSS rules that are purely for demonstration purposes;

technical sophistication is the skill you need to figure out that such demo code can safely be deleted, especially if a future listing omits it.

Later chapters, particularly after Chapter 5, also require that you successfully configure a *development environment* and run a templating system to build a professional-grade website. A huge number of things can go wrong when getting these to work; if you get stuck, there's no substitute for Googling around and determination—both of which are key aspects of technical sophistication.

The important thing at this point in your progress as a developer is to start learning how styling and layout concepts work together, how HTML and CSS combine to make a layout that is actually useful, and how to use some sort of system that allows you to avoid repeating sections of the site or styling on multiple pages (Box 1.3). Doing otherwise would make you an expert at changing text color and size, but you'd have no chance of applying the knowledge you've gained to a real-world scenario. In fact, by learning CSS in a holistic way, you'll not just be getting an introduction to styling—this tutorial is also going to be your first introduction to the world of front-end development (Figure 1.3).[4]

Figure 1.3: Yer a wizard front-end developer, Harry.

Box 1.3. Staying DRY

If you've been poking around the Internet in places where developers talk shop, you might have noticed someone mention staying *DRY*, with "dry" in all caps. They aren't talking about relative moisture levels. What they are talking about is a core principle in programming: *Don't Repeat Yourself*.

The idea behind DRY is that good coding should include as few instances of unnecessary repetition as humanly possible, simply because you have the same code in a bunch of places, then every time you want to make a change you'll have to update all the different spots in the application where that code is repeated. For example, if you wanted to change a link to the navigation bar of a hand-built site, you'd have to make the same change on *every* page. On a two-page site this wouldn't be a big deal, but for a bigger website it would be a nightmare.

Programmers are a special sort of lazy—especially when it comes to doing something repetitive that could be done more efficiently with a little bit of extra programming. To make it easier to be lazy, enterprising programmers spend countless hours creating systems that allow other developers not to have to repeat themselves. We all benefit from developers who at some point decided they were going to work really hard *now* so they could work less hard in the future.

Templating software, like the system we will be using in Chapter 5, allows us not to repeat ourselves by collecting repeated code into individual files, and then including these code snippets onto any page where they're needed.

The result is that we can write something like the navigation menu for a site *once*, put it in its own little file, and then include that file every place where the navigation needs to appear. If we want to change the navigation later on, we need to edit only that single file, and the changes will automatically be applied to every page that includes it. We'll learn how to do this in Section 5.6.

1.1.1 So, what is front-end development?

When someone says that they are a front-end developer, that means that they work on the parts of a site that people see and interact with. That includes things like HTML, CSS, and JavaScript. You'll also hear people talk about user interface (UI) design (the way things look) and user experience (UX) design (the way that the interface and different pages function to move users through the site to a goal).

The complement to front-end development is back-end development, which involves the architecture of data, how it is stored, and how it is delivered. Later Learn Enough tutorials (starting with *Learn Enough Ruby to Be Dangerous*) cover the basics of back-end development, culminating in the *Ruby on Rails Tutorial*, which teaches how to develop a full web application with a database, users, login and authentication, and more.

So, how do we turn an HTML caterpillar into a front-end developer butterfly (Figure 1.4)?

Figure 1.4: It's a whole lot easier to read Learn Enough CSS & Layout to Be Dangerous than to pupate.

We'll start by building on the styling introduced in *Learn Enough HTML to Be Dangerous*. In the final sections of that tutorial, we showed how to add styling directly to elements on a page, while also noting that such inline styling isn't the best way to go in practice. In this tutorial, we'll discuss the method that *is* the best practice, namely, CSS.

Throughout the rest of this chapter, we'll learn the basics of CSS declarations and values by starting with a few super-simple elements on a sample page, with a particular focus on applying the DRY principle (Box 1.3). We'll end with a first introduction to the essential technique of *CSS selectors* to target particular page elements for styling.

In Chapter 2, we'll discuss aspects of selectors that are important to get right at the beginning of a project, with a focus on managing complexity and maintaining flexibility by choosing good names for things (including an introduction to CSS color conventions).

Chapter 3 introduces two of the most important kinds of CSS values: colors and sizes. These lay an essential foundation for Chapter 4 on the *box model*, which determines how different elements fit together on the page.

In Chapter 5 and Chapter 6, we'll take the page that we've been working on and factor it into a layout using a templating system called *Jekyll* to build professional-grade websites that are easy to maintain and update.

In Chapter 7, we'll learn how to make flexible page layouts using *flexbox*, adding layouts for a photo gallery page (to be filled in in *Learn Enough JavaScript to Be Dangerous*) and a blog with posts. In Chapter 8, we'll add the blog itself, showing how to use Jekyll to make a professional-grade blog without black-box solutions like Wordpress or Tumblr.

Because a large and growing amount of web traffic comes from mobile devices, in Chapter 9 we'll cover the basics of using CSS and *media queries* to make mobile-friendly sites without violating the DRY principle (Box 1.3).

Finally, in Chapter 10 we'll add the kinds of little details (like custom fonts and `meta` tags) that make a site feel complete. The result will be an industrial-strength, nicely styled site deployed to the live Web.

Let's go!

1.2 CSS overview and history

CSS takes the form of plain text declarations inserted into an HTML or CSS file using a text editor. A typical series of CSS declarations might look like Listing 1.1. (You are not expected to understand these styles at this point.)

Listing 1.1: *Typical CSS declarations.*

```
body {
  color: black;
}

p {
  font-size: 1em;
}

p.highlighted {
  font-size: 1.5em;
  background: yellow;
}
```

The "Cascading" part of Cascading Style Sheets refers to the way the defined styles flow, or "cascade", down from element to element on a page based on a few factors like which declaration came first, whether an element is the child of a parent element that has styles applied to it, or the strength of the declaration (more on this in Section 2.3). This inheritance of style (from the top levels to elements below) happens so that we as developers can avoid having to define how every single element should look. For example, if we changed the `color` in the `body` tag in Listing 1.1, that change would cascade down and change the `color` attribute on every interior element as well.

The "Style Sheet" part of the name (sometimes written as the single word *stylesheet*) refers to how CSS allows developers to collect all the style declarations in a separate section of the page (called an *internal stylesheet*), or place them into an external file (called, you guessed it, an *external stylesheet*). External stylesheets are loaded onto the page as a link in the `head` section of the HTML. (We'll learn how to do this ourselves in Chapter 5.) The result is that we end up separating the code that defines how something looks (or is positioned) from the actual content—all of which makes for simpler and more maintainable code.

1.2.1 CSS always be changing

One important thing to note about CSS is that, like HTML, it's constantly evolving to better serve the needs of web designers and developers. In fact, in many ways CSS is evolving even faster than HTML (Figure 1.5).[5]

Figure 1.5: CSS… uh, finds a way.

Even though future additions are added to the official CSS specification all the time, they aren't evenly distributed—when new additions to CSS are proposed, the adoption of those new concepts happens on a browser-by-browser basis. A style that might work in Google Chrome might be totally unsupported by Mozilla Firefox or Microsoft Internet Explorer (or IE's latest incarnation, Microsoft Edge). Or it's possible that a given style might be supported, but only if you use a special temporary name to declare the style, a feature that lets developers target only the browsers that support the style they want to use (Box 1.4).

Box 1.4. Vendor Prefixing

Because it takes a while for an addition to the language to go from a suggested new specification (or *spec*) to an officially included part of CSS, new features aren't accepted by all browsers at the same time. Most browser makers aren't interested in waiting around for the spec to be officially updated, though—they want their software to push the envelope and do really cool stuff. So vendors take these CSS spec proposals and implement their own versions of the spec.

To get around the potential confusion that could happen if things work differently from browser to browser, the browser vendors typically add a prefix to the experimental styles, such as `-webkit-`, `-moz-`, and `-ms-` (respectively for WebKit, Mozilla, and Microsoft browsers). This allows the applied style to target specific browsers in case the support differs.

For instance, the CSS `transition` declaration (covered in Section 7.4) was implemented in most browsers before it was a part of the official spec, and to use it you would have needed to declare the styling like the following examples with vendor prefixes:

```
-webkit-transition: all 0.1s linear;
-moz-transition: all 0.1s linear;
-ms-transition: all 0.1s linear;
transition: all 0.1s linear;
```

The first rule here specifically targets browsers that use the WebKit layout engine (which includes Safari and Chrome), while the second targets browsers using Mozilla's Gecko engine (principally Firefox), and the third targets Microsoft browsers (Internet Explorer and Edge). Finally, the fourth rule is an unprefixed declaration—in this case, just `transition` by itself—which is included so that when support becomes official we won't have to go back into our code and add it in. (The `transition` style is supported by all major browsers today, so if you see the prefixed versions in code that you are working on, you can safely delete them.)

Luckily, at this point the most common styling definitions are essentially the same across different browsers,[6] and we aren't going to cover anything in this tutorial that has questionable browser support. At some point, though, you'll probably find yourself wanting to use a more cutting-edge style, and when that happens we recommend using a tool like CanIUse to figure out how well-supported the style is. Don't ever feel self-conscious about using reference sites like that—the fast-changing nature of the language and the spotty browser support make it a necessary tool even for people who have been doing this for years.

1.2.2 How did CSS develop?

In the beginning, content on the Internet was simply plain text. Then, as methods for organizing content started to come into existence (such as HTML around 1990), a number of methods sprang up to affect the appearance and layout of the data.

At first, many styling solutions that affected how the page looked would be set by individual users' browser preferences rather than by the creator of the page. As the complexity of the web increased, it became increasingly obvious that there should be a way for the *owner* of a site to at least suggest how a page should look, rather than leaving the appearance up to each individual browser.

Lots of interesting suggestions were put forward that never got widely adopted, usually because the proposed styling was overly complicated or used a totally non-intuitive structure.

- Robert Raisch developed RRP, which used arcane two-character style declarations and was fairly unreadable.
- Pei-Yuan Wei created the ViolaWWW browser and a styling system called PWP, which introduced nesting styles and external stylesheets, but it was only released for Unix operating systems and never really caught on.

- FOSI was created for an HTML precursor called SGML, and it worked by adding complicated tags on the page around content (not good).
- DSSSL allowed for complex declarations and was more of a programming language with styling attached, but it had a complicated syntax that made it overly complex for styling.

In short, there was a real Cambrian Explosion of proposed standards to make the web pretty, and while some of these systems ended up contributing elements to what became CSS, none of them are direct ancestors (Figure 1.6).[7]

Figure 1.6: There were a lot of precursors to CSS... not all of them successful.

About five years after the introduction of HTML, Håkon Lie (working with Bert Bos) put forward a styling system proposal in December of 1996 called CHSS (Cascading HTML Style Sheets). As you might expect, the original proposal has some details that are no longer present in the language, but in his doctoral thesis Håkon Lie simplified the specification into something that more closely resembles modern CSS. Eventually, the concept was adopted by the World Wide Web Consortium (W3C) as the system for styling web content.

Although it represented a big step forward, creating a specification is only half the battle—browsers have to support the standard for it to be of any use to end-user. No browser even partially supported CSS until 1997, and there wasn't full support of the standard in any one browser until March of 2000 (Figure 1.7)[8]—partially because browser makers still had their own ideas for how to accomplish styling, and partially because many browsers supported non-standard HTML tags (we're looking at you, Internet Explorer...).

Figure 1.7: After the party for the new millennium... we got CSS. Hooray?

Each browser handled CSS in its own idiosyncratic way for a long time, and anyone who has been working in front-end development for a while can tell you how maddeningly difficult it was to style a website so that it looked the same across browsers. Microsoft's Internet Explorer was *by far* the biggest problem (Figure 1.8) and we can only imagine the number of people who gave up on the world of web design and development thanks to horrific CSS support in early versions of IE.[9]

Figure 1.8: You can't fully understand hate until you try to design for IE6.

The differences between implementations of CSS in different browsers remained significant until just a few years ago, with the rise to dominance of the WebKit browsers (Google Chrome and

Apple Safari) and Gecko-powered Mozilla Firefox. So when reading this tutorial, just keep in mind that styling the web was a complete and utter mess until *very* recently—about 15 years after the initial release of the CSS spec in 1997.

1.2.3 The bog of eternal subjectivity

One last bit of meta-information before we jump in and start styling: deciding how to actually implement CSS on a given page can be a confusing mess. Whereas the previous Learn Enough tutorials (Box 1.1) were a little more prescriptive in nature, with many cases where there was only one right way to do things, we are now going to be venturing into places where there is often no right answer. When designing websites with CSS, many solutions to a problem typically exist, which means subjective judgment is the rule rather than the exception (Figure 1.9).[10]

Helping you navigate this mess is where we come in.

Figure 1.9: Smell baaaad!

Oh, and to make this all a little more fun, you may recall from Box 1.4 that every browser implements some parts of the CSS standard in its own slightly different manner, so you can never be absolutely certain that something will look the same to different users if they are using different browsers… to say nothing of how something will look on different screen sizes and resolutions for different operating systems (Chapter 9).

You have to get used to the idea that no site is going to be exactly the same when viewed by different people. You'll learn to design (or implement other people's designs) in a way that allows room for CSS's inherent ambiguity. Unlike the tightly constrained world of print design, getting things to look exactly the same in every browser and on every operating system is just something you have to give up worrying about.

Additionally, when talking about which CSS rules to use when styling a site or creating a layout, best practices tend to move in fads, or are influenced by the subjective opinion of the developers who worked on a project before you. For example, as discussed in Section 1.5, elements on the page frequently need to be assigned classes or ids, and the way that names are picked is entirely up to the person writing the code. As you might guess (if you know many developers or designers), people have *lots* of strong opinions on how you should be naming things—a classic holy war situation (Figure 1.10).[11]

Figure 1.10: HOLY WAAAAAAARS!

The most important thing is to be *consistent*. If you start a project and do things one way, make sure to keep following the same conventions for the life of the project. Or, if you decide to make a drastic change, spend the next couple of days updating all your old code, for the sake of future developers (including yourself). Keep an eye out in this tutorial for "Style Note" boxes that have tips on what the current best practices are for different uses of CSS.

1.3 Sample site setup

Now that we've got an overview of the purpose and origins of CSS, it's time to start looking at some concrete examples. The initial styling rules will necessarily be simple, so it's important to have patience while we lay this crucial foundation for the working website (with layout) that we'll be developing starting in Chapter 5.

We'll start by creating a new project in the `repos` folder using the same basic `mkdir` command covered in *Learn Enough HTML to Be Dangerous* (Listing 1.2).

Listing 1.2: *Adding the folder for our sample project.*

```
$ cd                                  # cd to the home directory
$ mkdir -p repos/<username>.github.io  # Make site directory
$ cd repos/<username>.github.io        # cd into new directory
```

Note that we've used a special directory name that corresponds to the main GitHub Pages site for your account:[12]

```
<username>.github.io
```

In Listing 1.2, `<username>` should be replaced with your GitHub username, so the full URL should look something like `mhartl.github.io`.

To get our site started, we'll also create an `index.html` file using the `touch` command (as discussed in *Learn Enough HTML to Be Dangerous*), as shown in Listing 1.3.

Listing 1.3: *Adding a blank* `index.html`.

```
$ touch index.html     # Create an empty index file
```

Inside the new folder, use your favorite text editor to open the newly created `index.html` file and paste in the markup shown in Listing 1.4. You should recognize this as a very basic HTML page—if this doesn't look familiar at all, we recommend reviewing *Learn Enough HTML to Be Dangerous* at this time.

Listing 1.4: *The initial HTML for our site.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>I'm an h1</h1>
    <ul>
      <li>
        <a href="http://example.com/" style="color: red;">Link</a>
      </li>
      <li>
        <a href="http://example.com/" style="color: red;">Link</a>
      </li>
      <li>
        <a href="http://example.com/" style="color: red;">Link</a>
      </li>
    </ul>
    <h2>I'm an h2</h2>
    <div style="border: 1px solid black;">
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div style="border: 1px solid black;">
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div style="border: 1px solid black;">
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div style="border: 1px solid black;">
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
```

```
  </body>
</html>
```

When you open that HTML document in your browser (Box 1.5), you'll see a series of three red links, some headers, and a series of green links in boxes (Figure 1.11). This will be our initial test page.

Note: Due to various slight differences between browsers, browser window sizes, etc., your results may not always match the screenshots exactly, so small discrepancies are not a cause for concern. As we'll emphasize throughout this tutorial, it's important to focus on achieving good-enough results without chasing the unreachable goal of pixel-perfection.

Figure 1.11: This is the beginning of something great, unassuming though it may seem.

Box 1.5. Opening HTML files

Depending on your exact setup, there are several ways you might open the HTML file shown in Listing 1.4. The most straightforward way at this stage is to use your native system, which is the main method discussed in *Learn Enough HTML to Be Dangerous*. On macOS, our favorite way to do this is with the `open` command, which opens the file using whatever the default program is for that file (e.g., for an HTML file it might use Safari or Chrome, for a PDF it might use Preview, etc.):

```
$ open index.html        # macOS only
```

The similar `xdg-open` command works on many Linux systems:

```
$ xdg-open index.html    # Linux only
```

You should be aware that using your native system will likely prove increasingly challenging as this tutorial progresses. The reason is that, Jekyll, the static site builder

we'll be using starting in Section 5.2, requires setting up your native system as a proper *development environment*, equipped with programming languages and other software needed to develop websites and dynamic web applications.

Our bet is that, if you're reading this tutorial, you're up to the challenge. This means taking the time to follow the steps in *Learn Enough Dev Environment to Be Dangerous* for your native system (either now or in Chapter 5).

There is one catch which might apply to some readers: there's no great solution if you're using Microsoft Windows. Indeed, one perennial challenge when teaching good development practices is the general difficulty of developing programs on Windows operating systems. As described in *Learn Enough Dev Environment to Be Dangerous*, for Windows users we recommend using either a *cloud IDE* or a *Linux virtual machine*. (In fact, the two cases are fundamentally the same, as the cloud IDE we recommend, Cloud9, runs Linux.)

The cloud IDE offers an especially convenient choice for getting up and running quickly, though one downside is that you need to be online to use it. If you do end up using the cloud IDE, opening an HTML file is as simple as clicking "Run" and then clicking the resulting URL to open it (Figure 1.12). (There are actually a couple of other methods that also work. Figuring out this sort of thing on your own is a perfect example of technical sophistication (Box 1.2).)

Eventually, though, it's a good idea to master your native system (especially if that system is Unix-based, e.g., macOS or Linux). Now's an excellent time to take that step.

Figure 1.12: Opening an HTML file using the cloud IDE.

As in *Learn Enough Git to Be Dangerous* and *Learn Enough HTML to Be Dangerous*, we'll deploy our new website immediately to production, which is a good habit to cultivate. First, you'll need to create a new project at GitHub, using the steps shown in *Learn Enough HTML to Be Dangerous*, which in this case looks like Figure 1.13.

Figure 1.13: Creating a new GitHub repo.

Once you've finished the steps needed to create the repository, initialize and deploy it using the commands shown in Listing 1.5.

Listing 1.5: *Deploying the initial site to GitHub pages.*

```
$ git init
$ git add -A
$ git commit -m "Initialize repo"
$ git remote add origin <repo url>
$ git push -u origin master
```

If you find the sequence of commands in Listing 1.5 challenging instead of familiar, it's probably a good idea to review *Learn Enough Git to Be Dangerous* and Starting the project from *Learn Enough HTML to Be Dangerous*. (Confirming that the deployment succeeded is left as an exercise (Section 1.3.1).)

Although we'll add in the occasional Git commit/deployment in this tutorial, in general these will be up to you. Having learned the material in *Learn Enough Git to Be Dangerous* and *Learn Enough HTML to Be Dangerous* (or already being familiar with it), now is a good time to practice making such decisions on your own.

1.3.1 Exercises

1. By visiting the URL <username>.github.io/index.html, confirm that the deployment to GitHub Pages succeeded. Is it necessary in include index.html in the URL?

1.4 Start stylin'

As discussed in Section 1.2, CSS is a way of defining how elements on an HTML page look and are positioned, with styling that flows ("cascades") down from element to element based on factors like which declaration came first, whether an element is the child of a parent element that has styles applied to it, or the specificity of the declaration.

So what do "parent", "children", and "specificity" mean in the context of a web page and how it gets styled?

The idea is that every element on the page is contained inside of another element, which in turn can contain other elements—like Russian nesting dolls (Figure 1.14).[13]

Figure 1.14: Russian dolls are designed to nest one inside the other.

We can visualize the parent and child structure of the elements on a typical page using the diagram in Figure 1.15.

Figure 1.15: Relationships always cascade down.

The hierarchical tag structure shown in Figure 1.15 is known as the *Document Object Model*, or DOM for short.[14] Each new level in the DOM is a child of the level above it. In other words, the `html` tag is the parent of the entire page, the `body` tag is a child of the `html` tag, and so on. The `body` tag then has its own children, which are the `h1` and `h2` elements, the unordered list `ul` , and the `div` elements. In CSS, styling rules flow down from parents to children unless another style interrupts and takes priority.

Our first application of these ideas will be to the repetitive inline styles that exist in our current example page. For example, several `div` elements share identical inline border styling, as seen in Listing 1.6.[15]

Listing 1.6: *Repeated inline styles violate the DRY principle.*

index.html

```
.
.
.
<div style="border: 1px solid black;">
<div style="border: 1px solid black;">
<div style="border: 1px solid black;">
.
.
.
```

Now that you know about the DRY principle (Box 1.3), all that redundant styling should be causing a mild case of programmer's itch, and the only cure is to eliminate the duplication. The technique we'll use is called *refactoring*, which involves changing the form of the code without changing its function. In other words, after making the changes in this section, the appearance in the browser should remain the same.

To make the example code cleaner, we are going to use a method mentioned in Section 1.2: an internal style sheet. To implement this, we need to use a new HTML element, the `style` tag (Listing 1.7). This `style` block will hold all our style declarations, but by design the `style` tag isn't visible on rendered pages, and thus won't be visible in a user's browser.

Listing 1.7: *The initial empty style block for our page.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>

    </style>
  </head>
  .
  .
  .
```

A `style` block like this can actually be placed anywhere on the page, but the conventional placement is inside the page's `head` tag. (The best practice is actually to put the CSS rules in a separate file, a task we'll undertake in Section 5.5.)

Now let's get rid of the repetition in Listing 1.6 by adding in our first CSS declaration, as shown in Listing 1.8.[16]

Listing 1.8: *Adding our first CSS style.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      div {
        border: 1px solid black;
      }
    </style>
  </head>
  .
  .
  .
```

Figure 1.16 shows the anatomy of the CSS rule from Listing 1.8: the `div` part of the statement, which is outside the curly braces, is called the CSS *selector* (in this case targeting only `div` HTML elements). Then there is a *declaration* made of up a property (`border`) and a *value* (`1px solid black`) separated from the property by a colon. Finally, there is a semicolon at the end of the line that ends the style. (Just a warning: a lot of these terms get mixed up in regular usage. For example, people will sometimes refer to the whole thing, including the selector, as the declaration.)

Figure 1.16: What's in a name?

Although the spacing shown in Listing 1.8 is typical, CSS is similar to HTML in that whitespace is ignored by the browser. For the sake of any humans viewing your markup, though, it's a good idea to follow certain formatting conventions (Box 1.6).

Box 1.6. Style Note: Formatting styles

The styling statement from Listing 1.8 can also be written with everything on one line, like this:

```
div {border: 1px solid black}
```

While that might look nice and tidy at first, you should probably avoid writing CSS that way, as it would make for unreadable code as you start to add more styles. It is far easier to read a list of style declarations like this:

```
button {
  background-color: gray;
  border: 1px solid black;
  color: white;
  cursor: pointer;
  display: inline-block;
  font-family: "proxima-nova", "Proxima Nova",
               sans-serif;
  font-size: 12px;
  font-weight: bold;
  letter-spacing: 0.15em;
  padding: 10px 15px;
  text-decoration: none;
  text-transform: uppercase;
  transition: all 0.1s linear;
}
```

than it is to read this:

```
button { background-color: gray; border: 1px solid
black; color: white; cursor: pointer; display:
inline-block; font-family: "proxima-nova",
"Proxima Nova",
sans-serif; font-size: 12px; font-weight: bold;
letter-spacing: 0.15em; padding: 10px 15px;
text-decoration: none; text-transform: uppercase;
transition: all 0.1s linear}
```

Now just imagine trying to find specific properties on a page with hundreds of styles where every declaration looks like that… "Nightmare" wouldn't even begin to describe your development experience.

A second point about formatting is that you'll notice that the style properties are all in alphabetical order. It might seem annoying to keep your properties alphabetized, but if you do you'll find that over time you will find things much faster than if the ordering is haphazard. Fortunately, your text editor of choice likely has an alphabetize feature. For example, in Sublime Text you can select multiple lines and hit F5 to automatically rearrange things in alphabetic order! Other editors frequently have packages that do the same.

After adding the CSS in Listing 1.8, delete all the `style="border: 1px solid black;"` attributes from the four `div` tags. Your test page's code should now look like Listing 1.9.

Listing 1.9: *How the entire page should now look.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      div {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <h1>I'm an h1</h1>
    <ul>
      <li>
        <a href="http://example.com/" style="color: red;">Link</a>
      </li>
      <li>
        <a href="http://example.com/" style="color: red;">Link</a>
      </li>
      <li>
        <a href="http://example.com/" style="color: red;">Link</a>
      </li>
    </ul>
    <h2>I'm an h2</h2>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
  </body>
</html>
```

Save the changes, refresh the page in your browser… and BAM! Everything should look the same. (If not, double-check your work to see if you can get your results to match.)

So what happened here? The declaration that we added in Listing 1.8 is a CSS statement that tells the browser it needs to apply a 1-pixel wide solid black border to all of the `div` s in the `body` of the `html` . (We'll learn more about pixels in Section cha:values_pixel.) The result is a simplification of the code without any change in the page's appearance.

Now that we've seen how to consolidate a bunch of inline styles into a single CSS declaration, let's do the same thing for the links that are colored red via inline styles inside of the `li` s. Your new sample page should look like Listing 1.10.

Listing 1.10: *Removing the red link colors from the HTML.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      a {
        color: red;
      }
      div {
        border: 1px solid black;
      }
    </style>
  </head>
  <body>
    <h1>I'm an h1</h1>
    <ul>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
    </ul>
    <h2>I'm an h2</h2>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/" style="color: green;">I'm a link</a>
    </div>
  </body>
</html>
```

As before, the appearance shouldn't change after refreshing the browser.

At this point, we've definitely made progress in the fight against inline style redundancy, but what about those links that are colored green at the bottom using inline styling? One way to clean up the inline styles would be to use *CSS classes*, which we'll start covering in Section 1.5, but for now let's see if we can do it with just generic CSS selectors.

Because the links in question are contained inside of `div` s, and the other links on the page are not, we can use the nesting inheritance of CSS to define a style that changes *only* the color of links inside of `div` s. We can accomplish this by adding the declaration from Listing 1.11 to the style block, and then remove the inline styles from the links on the page.

Listing 1.11: *Creating a style that targets links inside of* `div` *s.*

index.html

```
div a {
  color: green;
}
```

Your entire test page should now look like Listing 1.12.

Listing 1.12: *Whole page with new styling to target links inside divs.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      a {
        color: red;
      }
      div {
        border: 1px solid black;
      }
      div a {
        color: green;
      }
    </style>
  </head>
  <body>
    <h1>I'm an h1</h1>
    <ul>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
    </ul>
    <h2>I'm an h2</h2>
    <div>
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div>
      <a href="http://example.com/">I'm a link</a>
    </div>
  </body>
</html>
```

The page is a lot cleaner now, isn't it? Separating the styling from the content makes the page's code much easier to read, but maybe you can already see the problem with the way that we've targeted the styles...

The problem is: if we were to add new `div` s anywhere on the page, and they happened to have links inside them, those links would be green even if that isn't what we want. The reason why is that the selector we used in that declaration is way too generic. In Section 1.5, we'll take a look at how to add styling with far greater specificity using CSS ids and classes.

1.4.1 Exercises

1. Using what we learned about targeting links inside of other objects, change the color of the links inside of the `li` s.
2. Add a border around the `li` s using the same styling as we used to add the borders around the `div` s.

1.5 CSS selectors

Between *Learn Enough HTML to Be Dangerous* and the beginning of this tutorial, we have so far used only rudimentary targeting techniques for styles. In *Learn Enough HTML to Be Dangerous*, we learned how to add styles directly to elements, but this approach is brittle and inefficient. In this tutorial, so far we've used CSS that is separated from the content, but we have used only generic selectors like `div` or `a` . The problem with generic element selectors is that they apply to *all* the elements on the page. So, how can we apply styling to *specific* elements rather than to every single one?

There are two methods, one that targets only one element per page—the *id* (or "identification") selector[17]—and one able to target multiple elements—the *class* selector. Let's edit our example HTML to add this kind of targeting to our page.

Ids and classes are always applied only to the opening tag of an element, and they always have the same format. We'll use the `div` tag for concreteness:

```
<div id="foo" class="bar">
  .
  .
  .
</div>
```

We see here that both ids and classes consist of key/value pairs, where each value is a string that serves as a label for the id or class. In this case, the key `id` has value `"foo"` and the key `class` has value `"bar"` .

Although CSS offers a great deal of flexibility in choosing id and class names, there are a few restrictions and usage suggestions:

- Use only one id per element.
- No numbers are allowed at the beginning of the name (e.g., `name1` is valid, but `1name` isn't).
- Dashes (`-`), underscores `_` , and CamelCase can be used to join multiple words (so `foo-bar-baz` , `foo_bar_baz` , and `FooBarBaz` are all valid names).
- Spaces are invalid in id names, and are used to separate multiple names in the case of classes (so `id="foo bar"` is illegal, while `class="foo bar baz"` places three separate classes on an element).
- Be consistent (e.g., if using dashes as separators, use them everywhere—don't mix them with underscores).

To see how this works in practice, let's add some ids and classes to our sample page. On the first opening `div` tag that comes right after the `h2` , add `id="exec-bio"` , and then add `class="bio-box"` to all of the `div` s in that section, as shown in Listing 1.13).

Listing 1.13: *Adding CSS classes and an id to the sample page.*

index.html

```
<!DOCTYPE html>
<html>
  .
  .
  .
  <body>
    .
    .
    .
    <h2>I'm an h2</h2>
    <div id="exec-bio" class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
  </body>
</html>
```

(This use of a class is good CSS practice, but we generally don't recommend using an id in this context; we include it here mainly for demonstration purposes. We'll discuss this issue in more detail in Section 2.2.)

Next, let's update our CSS block to target these new selectors. To target an id in CSS, you put a `#` (usually read "hash") in front of the name, and to target a class you add a `.` (usually read

"dot"). For example, to change the background color of the `#exec-bio` id, we can use the following CSS rule:

```
#exec-bio {
  background-color: lightgray;
}
```

(Here `lightgray` represents (surprise!) a light gray,[18] which is an example of a *CSS color name*. We'll cover the details of color naming in Section 3.1.) Similarly, to apply a rule to the `.bio-box` class,[19] we can use the following CSS:

```
.bio-box {
  border: 1px solid black;
}
```

As we'll see in a moment, this rule keeps the thin black border added in Listing 1.9, but in such a way that it doesn't apply to *all* the divs on the site.

Finally, we can target the anchor tags inside the bio boxes using the combination of the class name and the tag name, like this:

```
.bio-box a {
  color: green;
}
```

This turns the `a` tags green, but only if they're inside an element with class `"bio-box"`. This class-based approach gives us much finer-grained control than the method used in Listing 1.11.

Adding these three rules to the `style` block (while removing the rules we no longer need) leads to the markup shown in Listing 1.14.

Listing 1.14: *Adding CSS rules to target the classes and id.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      a {
        color: red;
      }
      #exec-bio {
        background-color: lightgray;
      }
      .bio-box {
        border: 1px solid black;
      }
      .bio-box a {
        color: green;
      }
    </style>
  </head>
  <body>
    .
    .
    .
    <h2>I'm an h2</h2>
```

```
    <div id="exec-bio" class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
    <div class="bio-box">
      <a href="http://example.com/">I'm a link</a>
    </div>
  </body>
</html>
```

After saving your changes and refreshing the browser, you should see that the boxes at the bottom have the same border as before, but now the one with the CSS id now has a light gray background (Figure 1.17).

Figure 1.17: Elements targeted with classes and ids.

Congrats! You've just used ids and classes to target styles at specific elements, and have leveled up your CSS knowledge. Now that you've learned how to make declarations, and how to use ids and classes, we can now start getting into the nitty gritty of how CSS works.

1.5.1 Exercises

1. Try adding a new CSS id to the style section (you pick the name) that sets the background color of an element to `orange`, and then add that id to one of the links on the page.

2. Add a new class to the style section (you pick the name again) that changes the background color to `azure`, and add that class name as a second class on the `.bio-box` es. You'll notice that one box is different from the rest, something that we'll discuss in Section 2.3.

3. Commit your changes from this chapter and deploy the result to GitHub Pages. Confirm that the deployed site renders correctly. (It may take a minute or two for the cache to clear, so keep refreshing until the result is what you expect.)

Chapter 2
The style of style

We can think of CSS as operating at two main levels: the browser and the text editor. From the browser's perspective, the exact choices for CSS classes and ids is irrelevant; indeed, as far as the browser is concerned, there is hardly any difference between beautiful self-contained CSS with perfectly comprehensible class names and horrible inline styles on every element.[1]

At the level of the text editor, though, these concerns matter a lot to the people writing the HTML and CSS for the site—in this case, us. The browser might not care much about the repetition and complexity from inline styles and poorly named classes, but we sure do.

Moreover, bad styling choices can haunt us throughout a project, so it's important to do our best to get them right from the start (bearing in mind that we might have to make some changes down the line).

In this chapter, we'll focus on developing an understanding of the "style of style"—how to make good choices in naming and structuring the various parts of our site—as early as possible. The result will be a codebase that is flexible and maintainable, both for us and for any other developers who need to modify the site later on.

2.1 Naming things

As computer scientist Phil Carlton once remarked, "There are only two hard things in Computer Science: cache invalidation and naming things." This latter "hard thing" applies to front-end development as well.

When coming up with names for classes and ids, it's often helpful to think in terms of how something functions or what its intent is, and it's usually best to be specific. For example, making a class called `"box1"` is a bad idea because the name is so generic; on a big project, you might not remember what `"box1"` when you come back to the code at some point in the future. Better to introduce a class like `"bio-box"`, which makes reference to a specific kind of element on the page (in this case, a box for short biographies).

One important thing to avoid is naming classes or ids after how the element looks on the page. For example, suppose that for some reason the last `.bio-box` on the test page has information that we want a user to be alerted to, which we want to indicate by setting the background color of the box to red. We could add a class of `"red"` to the last box (as in Listing 2.1), making sure it is separated from the other class by a space, and then style it in the CSS like in Listing 2.2.

Listing 2.1: *Adding the class* `.red` *to a* `.bio-box`.

index.html

```
.
.
.
<div class="bio-box">
  <a href="http://example.com/">I'm a link</a>
</div>
<div class="bio-box">
  <a href="http://example.com/">I'm a link</a>
</div>
<div class="bio-box">
  <a href="http://example.com/">I'm a link</a>
</div>
<div class="bio-box red">
  <a href="http://example.com/">I'm a link</a>
</div>
.
.
.
```

Listing 2.2: *Creating the* `.red` *class styling.*

index.html

```
.
.
.
<style>
  a {
    color: red;
  }
  #exec-bio {
    background-color: lightgray;
  }
  .bio-box {
    border: 1px solid black;
  }
  .bio-box a {
    color: green;
  }
  .red {
    background: red;
  }
</style>
.
.
.
```

When you save your work and refresh the browser, you'll see that the background of the box has changed (Figure 2.1).

Figure 2.1: Sure, this worked. But is it a good idea?

But let's say that at some point in the future we decide that red isn't our favorite color for alerts anymore, and now we want to use purple. So we open up our project file, and in the CSS change the `background` property to `purple`, as in Listing 2.3.

Listing 2.3: *The problem when you name classes based on appearance.*

index.html

```
.
.
.
    <style>
      a {
        color: red;
      }
      #exec-bio {
        background-color: lightgray;
      }
      .bio-box {
        border: 1px solid black;
      }
      .bio-box a {
        color: green;
      }
      .red {
        background: purple;
      }
    </style>
.
.
.
```

Now the class name and the effect it has on the page not only don't match up, they are downright confusing (Figure 2.2).

Figure 2.2: Why do this to yourself?!

This might seem like no big deal on our simple test page, but imagine if that class were used on elements all over a project. We'd have two choices: go through and change all the class names on all the elements, or just live with the class and its effect being confusing.

Instead, if we use a naming convention where the class names are based on what the intended purpose is of the element on the page, and therefore used a more descriptive name like `"alert"`, then we can change the text color without needing to worry about contradictory or confusing names (Listing 2.4).

Listing 2.4: *A more appropriately named class based on the intention.*

index.html

```
.
.
.
    <style>
      a {
        color: red;
      }
      #exec-bio {
        background-color: lightgray;
      }
      .bio-box {
        border: 1px solid black;
      }
      .bio-box a {
        color: green;
      }
      .alert {
        background: purple;
      }
    </style>
```

```
.
.
.
    <div class="bio-box alert">
      <a href="http://example.com/">I'm a link</a>
    </div>
.
.
.
```

With the convention shown in Listing 2.4, if we later decide that alerts should be purple instead of red, there's nothing confusing in the code. Some other examples: instead of calling something `"small"`, choose a class name like `"collapsed"` if that is the functionality; or use `"disabled"` instead of `"gray"` for grayed out elements that a user isn't allowed to interact with.

Of course, there are exceptions, and ultimately the naming system is totally up to you, but as a rule of thumb, it's a good idea to stick with functional naming (Box 2.1).

Box 2.1. Style Note: Naming conventions

Strict prescriptive class naming systems have become popular recently, and with good reason: there were a lot of projects out there with naming conventions that were completely arbitrary—projects that had more in common with a Dr. Seuss story than a coherent development project. These sorts of strict naming systems are frequently used when there is no limit to the number of developers who might work on a project over its lifetime (think of web applications developed and managed at large corporations).

We aren't going to get into any of these in this tutorial, but we thought it would be a good idea to at least mention that they exist. If you get more into front-end development, it might not be a bad idea to look into some of the conventions other developers are using:

- Block Element Modifier (BEM)
- Object Oriented CSS (OOCSS)
- Scalable and Modular Architecture for CSS (SMACSS)

Whether you find these systems useful or not, the most important thing is to strive for some semblance of consistency.

2.2 When and why

One of the other decisions we have to make is to decide when we should use ids and when we should use classes. As noted in Section 1.5, ids are intended to target only one element on the page, while classes can target multiple things. To enforce this design, HTML elements will accept multiple class names on a single object (separated by spaces), but allow for only one id per element (anything after the first is ignored). But that isn't the whole story, because browsers

treat ids and classes differently, and here is the first place where we're going to run the risk of starting a holy war...

Our view is as follows:

> You should strive to use ids only when you absolutely have to (for example, if you are using JavaScript, and then use them *only* for JavaScript).

Yes, in Section 1.5 we added an id to the first `div` and targeted styles at it, but that was just for demonstration purposes, and in general it's a practice that should be avoided. The reason is that when you use an id to apply styles, it is nearly impossible to change that styling with another declaration without making your code full of ugly hacks.

To see why, take a look at Listing 2.5 and add our new `.alert` class onto that first `div` (the one that also has the id `#exec-bio`), while also changing the alert color back to `red`.

Listing 2.5: *Adding an alert class to an element with an id.*

index.html

```
<style>
.
.
.
.alert {
  background: red;
}
</style>
.
.
.
<div id="exec-bio" class="bio-box alert">
  <a href="http://example.com/">I'm a link</a>
</div>
<div class="bio-box">
  <a href="http://example.com/">I'm a link</a>
</div>
<div class="bio-box">
  <a href="http://example.com/">I'm a link</a>
</div>
<div class="bio-box alert">
  <a href="http://example.com/">I'm a link</a>
</div>
.
.
.
```

Save and refresh, and you'll notice that nothing changed; even though you'd expect the background to be red, it isn't, and instead looks just like Figure 2.1.

The problem is that ids are considered by the browser to have a higher *specificity*, and that means that any styles declared in the more specific statement will take precedence over less specific styles. You can think of classes like a machine gun spraying out lots of little projectiles, and ids like a rocket launcher. The style that gets launched by an id just has more power.

One way to make the `.alert` style apply would be to increase the specificity of our declaration by adding a new declaration that targets any element that has both the id `#exec-bio` and the class `.alert`, as shown in Listing 2.6.

Listing 2.6: *Overcoming the strength of an id style by combining the id and a class.*

index.html

```
.
.
.
<style>
  a {
    color: red;
  }
  #exec-bio {
    background-color: lightgray;
  }
  .bio-box {
    border: 1px solid black;
  }
  .bio-box a {
    color: green;
  }
  .alert {
    background: red;
  }
  #exec-bio.alert {
    background: red;
  }
</style>
.
.
.
```

The CSS in Listing 2.6 chains together the rules for an id and a class using `#exec-bio.alert`, and its effect in this case is to add a red background to the div from Listing 2.5 (Figure 2.3).

Figure 2.3: Combining the id and class makes for an unusually specific style.

Using a combined style selector like that will change the background color, but it will apply the style only in the specific situation where both that id and class are present. If your site relies on styles targeted at ids, then over time you'll find yourself adding more and more of these hyper-specific declarations. That isn't the most effective way to use CSS.

It's better in the long run to use a system that's modular, so that designing and developing the front-end of a site is like snapping together Legos. This way, you can add a class to an element and be confident that the styles from that class will be applied correctly.

In case you were wondering, the browser merges together all the different declarations and then sorts out conflicts property by property. This means that stronger declarations don't override *all* styles on an object, only the properties that are included in that stronger declaration. In the case of the `.bio-box` that we've been messing around with, all the properties in Listing 2.7 are being simultaneously applied.

Listing 2.7: *All the styles that are affecting the one* `.bio-box`.

```
#exec-bio {
  background-color: lightgray;
}
.bio-box {
  border: 1px solid black;
}
.alert {
  background: red;
}
```

```
#exec-bio.alert {
  background: red;
}
```

Through the magic of the browser, all those rules are merged, and conflicts are automatically resolved. In the case of Listing 2.7, the merged styling would look something like Listing 2.8 (which indicates unused rules with CSS *comments*, which are discussed further in Section 2.4).

Listing 2.8: *The merged styles with the superseded ones commented out.*

```
{
  /* background: red; */
  /* background-color: lightgray; */
  background: red;
  border: 1px solid black;
}
```

Because of the specificity of the `.alert` class, the `background-color: red` rule overrides both the general `background: red` and the more specific `background-color: lightgray` rules.[2]

Let's take a closer look at how the browser determines which rules take precedence.

2.3 Priority and specificity

CSS was designed to allow for multiple stylesheets from multiple locations to influence the appearance of a single document without catastrophically crashing. The result is a system of priority and specificity rules devised to resolve contradictory style declarations like the ones we were playing with in Section 2.2.

To make this clearer, let's take a look at the simplified example shown in Listing 2.9, which uses a `width` set to a particular *percentage*. (We'll learn more about percentages in Section cha:values_percent.)

Listing 2.9: *Different rules targeting the same class.*

index.html

```
.
.
.
<style>
  .
  .
  .
  .bio-box {
    width: 75%;
  }
  .bio-box {
    width: 50%;
  }
</style>
.
.
.
```

If you update the style section of your test page with the styling above, you'll notice that the boxes end up being half of the width (50%) of the page—i.e., the second rule in Listing 2.9 is the one that gets applied (Figure 2.4). This is part of a general pattern: in the case of conflicting CSS rules, the final one gets applied.

Figure 2.4: A 50%-width box showing how the final CSS rule gets applied.

The full list of CSS priority rules appears in Table 2.1. You don't need to memorize this table—over time you'll get a feel for how the priority works. Also, of all these rules, only numbers 3 and 5–8 are priorities that you'll have to understand, and you should *strenuously* try and avoid using numbers 1 (Box 2.2) and 2 (makes for difficult-to-maintain code). Numbers 4 and 9 are out of your control.

1	Importance	Adding `!important` (e.g., "`width: 100% !important`") to a value overrides all other similar styles (but never use `!important` (Box 2.2)
2	Inline	A declaration that is put on an element using `style=`
3	Media Type	When a style is applied through a media query (more in Chapter 9)
4	User defined	Most browsers have the accessibility feature: a user defined CSS
5	Selector specificity	Styling applied via a class or id overwrites generic styling

6	Rule order	The last style written has priority
7	Parent inheritance	If there is no style specified, then children inherit styles from their parent
8	CSS	CSS rules from a stylesheet or `style` block that are applied to generic elements.
9	Browser defaults	Lowest priority, these are the default styles that browsers ship with

Table 2.1: CSS priority rules.

Now you might be asking, "OK, but now what happens when applying two styles that have the same priority?" In this case, we need to consider *specificity* as well as priority, which is used to resolve any situation where multiple styles with the same priority level are applied (#5 in Table 2.1).

At the most basic, specificity just means that the more specific you are when you target an element, the greater the strength the browser will give to the styles in that declaration. For example, suppose we wanted to make all `a` elements gray by declaring a style as in Listing 2.10.

Listing 2.10: *A not-very-specific style.*

```
a {
  color: gray;
}
```

Without even needing classes or ids, we could override this style by being a little more specific. So if we had links that were inside of `h1` headers, we could make all `a` s that are inside of those `h1` elements green using a declaration like the one in Listing 2.11.

Listing 2.11: *A more specific style.*

```
h1 a {
  color: green;
}
```

That slightly more specific declaration overrides the initial styling that made the `a` text gray, and instead the text would be green. (This is the technique we used in Listing 1.11).

Table 2.2 has a more detailed listing of the values that different selectors are assigned by the browser. The styles get more specific as you go down the table, meaning that lower ones would override the styles above.

You'll notice that there isn't a simple numbering system for Table 2.2 like there is for the priority list in Table 2.1. That's because the specificity uses a separate system that has different levels noted by adding new numbers with a comma—it's confusing and we've never actually heard of a developer work through a specificity problem using the number system, but we've included it for the sake of completeness.

Simple HTML selector	`em {color: #fff;}`	1
HTML selector targeting element inside another element	`h1 em {color: #00ff00;}`	2
CSS class name	`.alert {color: #ff0000;}`	1,0
HTML element with a class name	`p.safe {color: #0000ff;}`	1,1
CSS id	`#thing {color: #823706;}`	1,0,0
CSS id with a class name	`#thing .property {color: #823706;}`	1,1,0
Inline style	`style="color: transparent;"`	1,0,0,0

Table 2.2: The confusingly complex rules of specificity.

This all seems really complicated, right?

Well, most developers don't actually know all these rules by heart. Instead, what we all use are systems of simple conventions like trying to keep declarations simple and general, with exceptions targeted via a class, not using ids to target styles, not using `!important` (Box 2.2),[3] etc. Over time you'll build up a gut feeling for the specificity.

Box 2.2. Style Note: Never use `!important`

There's another bad way of getting a style to apply, and that is to use the `!important` flag to the declaration, which automatically overrides any conflicting styles (Table 2.1). You should think about `!important` (read "important"; the "!" is silent) like this: if you've had to use `!important`, then you've failed at styling something.

The problem with using `!important` is that once you start using it, there's a tendency to use it more and more over time, since the only way to overcome a style that was applied with `!important` is to use another `!important`. Such proliferating `!important` rules are the tribbles of CSS.

It is always better to rethink how you are styling something than to use `!important`.

Now that you know about `!important`, erase it from your memory.

One way to avoid getting caught in situations where overlapping layers of complexity keep your styles from being applied is to try to keep your selectors as simple as possible (Listing 2.12), rather than using an ugly and complicated set of selectors (Listing 2.13).

Listing 2.12: *Good clean CSS.*

```
.bio-box a {
  color: green;
}
.alert {
```

```
  background: red;
}
```

Listing 2.13: *Ugly and complicated CSS.*

```
body div#exec-bio.bio-box a {
  color: orange;
}
```

Often simplicity is the best solution.

2.3.1 Exercises

1. Use the `!important` flag to force the background color of the `.alert` class to be red (make sure to remove `#exec-bio` from the style).

2. Remove what you did in Exercise #1 and promise to never use `!important` again (Box 2.2).

3. Try changing the color of the `.bio-box a` links not by changing the `color` property on the existing style, but instead by adding a new identical selector below the existing one and a new `color` declaration that changes the link color to `pink`.

2.4 How to be a good styling citizen

So, how should you be a good developer and use CSS selectors in a reasonable way that takes advantage of the crazy priority and specificity rules... without requiring a lot of mental overhead? Start with the concept we mentioned in Section 2.2: that classes should be combined like Legos to get the result we are looking for. Make design choices that are modular so that your styles only affect things inside of modules instead of leaking out to affect elements site-wide.

If you need to have a module do slightly different things depending on placement or status, multiple classes on an element are a valid usage of the class selector, but at the same time don't take that to mean that you need to give every single element on the page a class, or (even worse) multiple classes. It's a fine balancing act between under- and over-classing your markup.

Looking at the styling we've done, you might have wondered why we didn't give the links in the `.bio-box` es classes of their own. We definitely could have, and there would have been nothing wrong with that, but this is another one of those subjective areas in styling. One good practice is to divide up the styling into two different categories: global styles that will apply in many different places in order to create greater consistency, and individual sections that are self-contained modules of functionality or content.

As an example, let's assume that the `.bio-box` es are going to be located in a part of the site that always has one link in each box—they are going to be repeating and regular modules. In this

case, we can just skip classing the individual links and apply a more general style for an `a` tag within the `.bio-box`, without needing to drill down and add new styles. But if we need to add another link, then we'd have to consider whether or not we want the two links to look the same. If not, it would be necessary to come up with a different way to target the elements.

Let's add some content to make these ideas more concrete. Replace the dummy `.bio-boxes` with the markup shown in Listing 2.14, which includes the entire page to help you sync up. Note that, among other things, we've eliminated the `exec-bio` id, which (as mentioned in Section 2.2) we recommend using only when absolutely necessary, and we've also slimmed down the CSS rules.

Listing 2.14: *Using more realistic example HTML.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      a {
        color: red;
      }
      .bio-box {
        border: 1px solid black;
      }
      .bio-box a {
        color: green;
      }
    </style>
  </head>
  <body>
    <h1>I'm an h1</h1>
    <ul>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
    </ul>
    <h2>I'm an h2</h2>
    <div class="bio-box">
      <h3>Michael Hartl</h3>
      <a href="http://twitter.com/mhartl">here</a>
      <p>
        Known for his dazzling charm, rapier wit, and unrivaled humility,
        Michael is the creator of the
        <a href="http://railstutorial.org/">Ruby on Rails
        Tutorial</a> and principal author of the
        <a href="https://learnenough.com/">
        Learn Enough to Be Dangerous</a> introductory sequence. Michael
        is also notorious as the founder of
        <a href="http://tauday.com/">Tau Day</a> and author of
        <a href="http://tauday.com/tau-manifesto"><em>The Tau
        Manifesto</em></a>, but rumors that he's secretly a supervillain
        are slightly exaggerated.
      </p>
    </div>
    <div class="bio-box">
      <h3>Lee Donahoe</h3>
```

```
      <a href="https://twitter.com/leedonahoe">here</a>
      <p>
        When he's not literally swimming with sharks or hunting powder stashes on
        his snowboard, you can find Lee in front of his computer designing
        interfaces, doing front-end development, or writing some of the
        interface-related Learn Enough tutorials.
      </p>
    </div>
    <div class="bio-box">
      <h3>Nick Merwin</h3>
      <a href="https://twitter.com/nickmerwin">here</a>
      <p>
        You may have seen him shredding guitar live with Capital Cities on Jimmy
        Kimmel, Conan, or The Ellen Show, but rest assured Nick is a true nerd at
        heart. He's just as happy shredding well-spec'd lines of code from a tour
        bus as he is from his kitchen table.
      </p>
    </div>
    <div class="bio-box">
      <h3>??</h3>
      <p>
        The Future
      </p>
    </div>
  </body>
</html>
```

If you save and refresh the page, you'll find that all of the links in the boxes look the same, i.e., they're green (Figure 2.5).

Figure 2.5: All-green links.

Looking back at Listing 2.14, we see that each bio contains a link to the corresponding person's Twitter account, and it would be nice to distinguish these visually from the others. We'll continue

to improve these links throughout the tutorial, but for now we'll just make them blue, while keeping the others green.

One way to do this would be to retarget the style that makes the links green by making the style only apply to links that are *inside* of paragraph `p` tags, as shown in Listing 2.15.

Listing 2.15: *Styling link color inside of the paragraphs.*

index.html

```
.bio-box p a {
  color: green;
}
```

This would change the Twitter links to red (the color specified by the generic rule for anchor tag in Listing 1.10), reserving green for links inside paragraph tags, but it would also start bumping up against a suggested *three selector limit* rule of thumb, as discussed in Box 2.3.

Box 2.3. Style Note: Selector depth

In general, and for a couple of reasons, it's a good idea to keep the number of selectors in a declaration under three (i.e., to three or fewer). This might seem easy right now with our super-simple test page, but on a complicated site things can get very messy. One obvious reason you want your selectors to be as short as possible is for readability. It's just easier to find what you are looking for in a big section of CSS if the selectors are short.

The other reason is that CSS selectors are read by the browser from right to left, so the more selectors there are, and the more general they are, the more work the browser has to do to render your page. It's a little counter-intuitive, since you'd think that the browser would start on the left and narrow down the scope of the styling by moving right... but for technical reasons it doesn't. So if you declared a style using `#first-table tr td h1`, the browser would first identify all `h1` s, then all `td` s, then all `tr` s, and finally restrict everything to just the elements that are in something with an id of `#first-table`.

If you have lots of elements on a page, this kind of inefficiency can really slow down rendering times, so keeping the number of selectors down is good both for us (the developers) and for our users.

Using the rule shown in Listing 2.15 would be fine—we'd just have to pay attention and avoid making it more complicated in future changes—but let's instead use a more robust practice by applying the specificity rules to get what we want while still adhering to Box 2.3's three-selector limit. We can accomplish this by changing the style selector back to just `.bio-box a` while adding a class of `.social-link` to all the Twitter links (Listing 2.16).

Listing 2.16: *Adding a class to the social media links.*

index.html

```
.
.
.
<a href="http://twitter.com/mhartl" class="social-link">here</a>
```

```
.
.
.
<a href="http://twitter.com/leedonahoe" class="social-link">here</a>
.
.
.
<a href="http://twitter.com/nickmerwin" class="social-link">here</a>
.
.
.
```

Then we can style the links by adding the new class declaration in the CSS, as in Listing 2.17. (Put this new declaration somewhere below the `.bio-box a` declaration.)

Listing 2.17: *Adding a style declaration for our social links.*

index.html

```
.
.
.
a {
  color: red;
}
.bio-box {
  border: 1px solid black;
}
.bio-box a {
  color: green;
}
a.social-link {
  color: blue;
}
.
.
.
```

Now, any links that are inside of paragraphs will be green, while the social links will be a nice blue (Figure 2.6).

Figure 2.6: The class combined with `a` *allows the styling to apply.*

When combined, the class and element selectors have higher specificity than `.bio-box a`. So, if you were to remove the `a` from `.social-link` in the selector, the link would turn green again.

So what's the point of this seemingly simple exercise?

It wouldn't seem like choosing a link color is all that important, but little mistakes in how you deal with specificity at the beginning of a project can cause problems down the road. Even in this simplified example, there are a bunch of decisions that have to be made that could affect the future development of the page. Choosing poorly could require us to have to go back to the code and rewrite it if our original styling was too generic or restrictive.

For example, suppose we kept the green link styling targeted using `.bio-box p a`. If we then later wanted to put images into the links—along with styles that target the images—we would have to add classes to all the images to cleanly target them (because `.bio-box p a img` is too many selectors deep). On the other hand, while classes for the images might be a good option if there are only a few, if there are a whole lot of images it could be a hassle to add in class names on each and every one.

You could get around both of these problems by adding a class to the `p` tag, which would let you cut out a level in the declaration, but then what happens if you want multiple paragraphs in a

bio? Now each `p` would need a class... and you are back into the mess of having to add class names to a large number of elements. An easy solution would be to wrap that whole text section in a new `div` with a class like `.bio-copy`, and then target the links inside with `.bio-copy a` and images in links with `.bio-copy img`.

Let's look at an example of what it looks like to wrap the text content in an element with its own class to allow for more precise targeting. This involves adding `.bio-copy` wrappers around each biography's copy,[4] as shown in Listing 2.18, which we'll put to use in a moment.

Listing 2.18: *Wrapping each of the bios in a `div` to better-target text.*

index.html

```
.
.
.
<div class="bio-box">
  <h3>Michael Hartl</h3>
  <a href="http://twitter.com/mhartl" class="social-link">here</a>
  <div class="bio-copy">
    .
    .
    .
  </div>
</div>

<div class="bio-box">
  <h3>Lee Donahoe</h3>
  <a href="https://twitter.com/leedonahoe" class="social-link">here</a>
  <div class="bio-copy">
    .
    .
    .
  </div>
</div>

<div class="bio-box">
  <h3>Nick Merwin</h3>
  <a href="https://twitter.com/nickmerwin" class="social-link">here</a>
  <div class="bio-copy">
    .
    .
    .
  </div>
</div>
.
.
.
```

Now let's look at another styling wrinkle. If you were to stick with the style declaration `a.social-link` from Listing 2.17 as the way to target all `.social-link` s, then you'd have to ask yourself, "Could there be a situation where I want the styling of a `.social-link` on an element that isn't actually a link?" It sounds a little weird to ask the question, since it would seem that if you call something a link, you'd expect it always to be a link, but you'll often have styling for links that you need to apply to something that isn't an `a`.

An example of this would be if you have a navigational menu where you have a bunch of links to pages. Often, you want to have a menu item for the current page a user is looking at, and you

don't want users to be able to click a link for the page they are already on (that would refresh the page), but you *do* want this type of menu item to look like the rest of the navigational links. In that case, the menu item that isn't a link would need to inherit all the same styling that the links got, and in our example that would be difficult if the selector was a combined HTML element and class name: `a.social-link`.

To allow the `.social-link` class to change the styling of the link without combining the HTML element and the class name, we should retarget the generic declarations in favor of the more specific method using the `.bio-copy` class name from Listing 2.18. The result is a change from the current `.bio-box` rule from Listing 2.12 to the more targeted style, which affects only links that are in the `.bio-copy` section of the boxes:

```
.bio-copy a {
  color: green;
}
```

At this point, we can also change `a.social-link` to just `.social-link`:

```
.social-link {
  color: blue;
}
```

Putting these together gives the final CSS rules for this chapter, shown in Listing 2.19.

Listing 2.19: *Applying good class and style practices.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      a {
        color: red;
      }
      .bio-box {
        border: 1px solid black;
      }
      .bio-copy a {
        color: green;
      }
      .social-link {
        color: blue;
      }
    </style>
  </head>
  <body>
    .
    .
    .
  </body>
</html>
```

Because the CSS changes we've just made were a refactoring, the appearance should be the same as before (Figure 2.6).

As a final step, we'll add some descriptive *CSS comments* while rearranging our CSS rules so that they are grouped thematically, according to whether they're global, apply only to social links, or apply to the page biographies. This practice makes it easier to navigate, read, and edit the CSS rules later on (Box 2.4). The result appears in Listing 2.20, which shows the full page in case you need to sync up.

Listing 2.20: *The final form of our page for this chapter.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      /* GLOBAL STYLES */
      a {
        color: red;
      }

      /* SOCIAL STYLES */
      .social-link {
        color: blue;
      }

      /* BIO STYLES */
      .bio-box {
        border: 1px solid black;
      }
      .bio-copy a {
        color: green;
      }
    </style>
  </head>
  <body>
    <h1>I'm an h1</h1>
    <ul>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
      <li>
        <a href="http://example.com/">Link</a>
      </li>
    </ul>
    <h2>I'm an h2</h2>
    <div class="bio-box">
      <h3>Michael Hartl</h3>
      <a href="http://twitter.com/mhartl" class="social-link">here</a>
      <div class="bio-copy">
        <p>
          Known for his dazzling charm, rapier wit, and unrivaled humility,
          Michael is the creator of the
          <a href="http://railstutorial.org/">Ruby on Rails
          Tutorial</a> and principal author of the
          <a href="https://learnenough.com/">
          Learn Enough to Be Dangerous</a> introductory sequence. Michael
          is also notorious as the founder of
          <a href="http://tauday.com/">Tau Day</a> and author of
          <a href="http://tauday.com/tau-manifesto"><em>The Tau
          Manifesto</em></a>, but rumors that he's secretly a supervillain
          are slightly exaggerated.
        </p>
      </div>
    </div>
```

```
    <div class="bio-box">
      <h3>Lee Donahoe</h3>
      <a href="https://twitter.com/leedonahoe" class="social-link">here</a>
      <div class="bio-copy">
        <p>
          When he's not literally swimming with sharks or hunting powder
          stashes on his snowboard, you can find Lee in front of his computer
          designing interfaces, doing front-end development, or writing some of
          the interface-related Learn Enough tutorials.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <h3>Nick Merwin</h3>
      <a href="https://twitter.com/nickmerwin" class="social-link">here</a>
      <div class="bio-copy">
        <p>
          You may have seen him shredding guitar live with Capital Cities on
          Jimmy Kimmel, Conan, or The Ellen Show, but rest assured Nick is a
          true nerd at heart. He's just as happy shredding well-spec'd lines
          of code from a tour bus as he is from his kitchen table.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <h3>??</h3>
      <p>
        The Future
      </p>
    </div>
  </body>
</html>
```

Box 2.4. Style Note: Group your style and add comments!

It might seem obvious, but if you have any concern for other human beings who might ever look at your code, for the love of `$DEITY`, please group all styles that relate to the same part of the site in the same place, and to be extra helpful add in a comment or two that explains what the styles are for!

CSS comments are any text that is between the asterisks `/* */`, which looks like this in practice:

```
/* HOMEPAGE STYLES */
```

Comments don't affect how the code is displayed to users, but you should know that they can be seen by anyone who browses the site source code... so don't put anything into the comments you'd be embarrassed to say out loud in a crowded place.

From this point on, when we add new styles we'll usually include a comment name for the section, like the `/* HOMEPAGE STYLES */` above. If the styles should be grouped into an existing section, we'll indicate that using this convention:

```
/* HOMEPAGE STYLES */
.
.
.
.some-style {
}
```

That means that you should put the new styles somewhere after the existing styles in that section (represented by the vertical ellipsis). The only time that we won't include the section name when adding styles is when we are already working and making changes in that section.

2.4.1 Exercises

1. Add a new style that sets generic `div` s to have a border style of `border: 1px solid green` . Save and refresh and all `div` s other than the `.bio-box` es should have a green border. Change the selector to `div.bio-box` , and then save and refresh.

2. Add the `.social-link` class onto the `h1` . Even though it isn't a link, the color should change.

3. Add your own comment to the style section, and inside add `html{background:red;}` . Save and refresh. Then delete the first `/*` and then save and refresh. Your page should look very different—always remember to balance your comment tags.

Chapter 3
CSS values: color and sizing

Now that we've learned how to make our site's skeleton, it's time to start fleshing it out with some more CSS values. In this chapter, we'll learn about two of the most important kinds of values that CSS can apply to HTML elements: color and sizing. These will allow us to go from putting elements on a page (Chapter 2) to controlling the color of the elements and how big they are.

Values in a CSS declaration (Figure 1.16) can take a lot of different forms, from numbers, to dimensions, to idiosyncratic options, to colors, etc. On top of all of that, there are shorthand methods that let you write multiple style properties and values on a single line. Most CSS declarations and values are self-explanatory—not too many people are going to be confused by `text-align: left` —but there are quite a few that have extra complications, weird exceptions, or just odd ways of defining a value.

The next few sections are going to recap some style values that you might have seen before, but we'll also dive into some of the less obvious use cases.

3.1 CSS color

So far in this tutorial we've defined colors with descriptive words like `red` , `green` , and `lightgray` . CSS supports a large number of such *color names*, and there are online references that list all the color names supported by all browsers. It's not the most flexible or even the most common system of defining CSS colors, though, and in this section we'll discuss other more powerful ways of applying colors in CSS.

3.1.1 Hexadecimal colors

As discussed previously in *Learn Enough HTML to Be Dangerous*, one common method for defining colors is *hexadecimal RGB* (red-green-blue). While this name might sound complicated, in practice the concept is fairly simple.

A quick way to show how hexadecimal color works is to change the color of the red link text on our sample page to its equivalent hexadecimal RGB color. Change the word `red` in the color property to the one shown in Listing 3.1.

Listing 3.1: *Switching from color name to a specific color value.*

index.html

```
/* GLOBAL STYLES */
a {
  color: #ff0000;
}
```

Now when you save and refresh your browser, the link text will still be the same bright red (Figure 3.1).

Figure 3.1: Sometimes no change is good change.

The reason the color system is called *hexadecimal* RGB is that it uses base-sixteen numbers instead of the usual base ten ("hexadecimal" is a mishmash of Greek and Latin meaning "six" (hex) and "tenth" (decimal)). In hexadecimal, or *hex* for short, `0` is equal to 0, and `f` is equal to 15—letting you count 16 values in a single numeral (Table 3.1).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Table 3.1: Counting in hex.

In base ten, we can count from 0 to 99 with two digits, where $99 = 10^2 - 1$. Similarly, hex lets us count from 0 to ff = $16^2 - 1 = 255$. In other words, putting two hex numbers next to each other lets us count from 0 to 255 using just two characters, with `00` = 0 and `FF` or `ff` = 255 (CSS hex is case-insensitive, so it doesn't matter if you use upper- or lower-case letters).

A computer monitor consists of picture elements, or *pixels*, and displays colors by combining the light from red, green, and blue elements of a pixel (Figure 3.2). Hexadecimal RGB puts three sets of two hex numbers next to each other to define the red, green, and blue values that make up a single color, so `#ff0000` can also be read as `red=ff, green=00, blue=00` or `red=255, green=0, blue=0`. As we mentioned in Listing 3.1, this is the same as the color word `red`.

Figure 3.2: The elements in a single pixel of a computer screen's display.

If all three colors are turned on (each set to `ff`, or `#ffffff`), the pixel will look white; if they are all off (each set to `00`, or `#000000`), it will look black. Combinations of the three colors can be used to create all the colors you see (Figure 3.3).

Figure 3.3: Some hexadecimal color examples.

CSS also supports a useful shorthand in the common case that some of the hex numerals are the same. If the digits are the same, as in `#222222`, `#cccccc`, or `#aa22ff`, we can shorten the whole number to just three digits, like this: `#222`, `#ccc`, or `#a2f`. When the browser sees only three digits, it fills in the missing ones.

Thus, using `#f00` in place of `#ff0000` (Listing 3.1) should lead to the same color red (Section 3.1.3). The result of making this change on our sample site is shown in Listing 3.2. The result should be the same as the previous version shown in Figure 3.1.

Listing 3.2: *Using the more compact hex notation.*

index.html

```
.
.
.
/* GLOBAL STYLES */
a {
  color: #f00;
}
.
.
.
```

The RGB color system might seem confusing at first, but with practice you'll quickly come to understand how the three values work together to make different colors, and different shades of those colors. For making more complicated colors, we suggest using a *color picker*, but there are some common cases that you should know off the top of your head.

For instance, the grayscale spectrum from black to white always has all three hex numbers the same: all `00` (or `#000000`) is black, all `ff` (or `#ffffff`) is white, and numbers in the middle, like `#979797`, are some shade of gray (Figure 3.4).[1]

Figure 3.4: Hexadecimal grayscale.

In practice, most web developers and designers use common hex values interchangeably with their color-word equivalents, so it's important to know that `#000` and `#000000` are both `black`, `#fff` and `#ffffff` are both white, `#00f` and `#0000ff` are both `blue`, etc.

3.1.2 Setting color and transparency via rgb() and rgba()

In addition to using RGB hex, you can also use RGB directly using `rgb()`, which allows you to use decimal numbers in place of hex. In other words, `rgb(255, 255, 255)` is the same as `#ffffff`, etc. But the main reason to use RGB directly is to set *transparency* via the `rgba()` command.

In `rgba()`, the `a` stands for *alpha*, because the conventional name for transparency level in image processing is the *alpha level*. The alpha level is indicated using a number between 0 and 1, where 0 is transparent, 1 is opaque, and decimals in between define all the levels of partial transparency (50% is 0.5, 25% is 0.25, etc.).

For example, let's make the social link's background a transparent gray using `rgba()`. We'll select a fairly dark gray, corresponding to RGB values of 150 each (out of 255), and initially set an opacity of `1` (Listing 3.3).

Listing 3.3: *Using the `rgba()` property to set an opaque background color.*

index.html

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 1);
  color: blue;
}
.
.
.
```

The result appears in Figure 3.5.

Figure 3.5: Social links with an opaque gray background.

Now let's switch to 50% opacity (which is the same as 50% transparency), as shown in Listing 3.4.

Listing 3.4: *Using the* `rgba()` *property to set a partially transparent background color.*

index.html

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 0.5);
  color: blue;
}
.
.
.
```

As you can see by comparing Figure 3.5 with Figure 3.6, the social links now have a partially transparent gray background. (We'll see another more practical example of transparency in Section 5.7.1.)

Figure 3.6: Blue text and 50% transparent gray background.

Amazingly, there are still more ways to set colors in CSS (HSL and HSLa), but we aren't going to go into them, as you are far less likely to encounter them in practice.

3.1.3 Exercises

1. Use a color picker, change the color of the links on the page (the `a` declaration) to a light purple color using a hex value.
2. Convert remaining color words to their hex equivalents, using the more compact notation if possible.
3. Use the `rgba` color system to fade the links by making their opacity 20%.
4. Set the links that are inside of the `li` s at the top of the page to have a class of `.social-link` .

3.2 Introduction to sizing

In both *Learn Enough HTML to Be Dangerous* and in this tutorial, we frequently use *pixel* measurements to set the sizes of things like fonts, margins, and padding (Chapter 4), but there are actually a confusingly wide variety of different ways to define the size of elements. As a result

of having to reliably display multiple generations of HTML across many different computers and devices, web browsers are an unimaginably complicated kluge of different standards. No one sat down and planned out how the web would work in one go—features and support aggregated over time like a snowball rolling downhill, as various people suggested new methods of styling. Dimensional units are one area that has seen many such additions over time.

You might think it would be easiest for us to do what we've been doing and almost always specify sizes using pixels (`px`)—after all, isn't a screen just a big grid of pixels?

Sizing an element this way would work great if everyone in the world had exactly the same screen size and resolution, but they don't, and some screens combine many physical pixels into a smaller number of virtual pixels. That means that when you size something to look good on your screen, someone using a lower resolution screen might find the element size to be unusably large, or if they are using a high-resolution screen (like a Retina display on an iPhone or iMac) it would look really small (Figure 3.7).[2]

Figure 3.7: Different-sized displays have wildly different pixel densities.

The good news is that modern browsers can zoom in and out to make things easier, but that can cause page layouts that use absolute sizes to break. Also, modern devices can scale their output to make something that has an incredibly high pixel density work as though it were a normal lower density screen.

For most of the last few years, the agreed-upon modern best practice has been to mostly use *relative* sizes, which style element sizes based on the size of other elements, or even to the size of the screen. For example, for a header you might say (in CSSese), "I want my header text to be four times as big as the default text size", and the browser would figure out the rest.

This type of relative sizing helps deal with the different screen scale issues, and also allows for easy resizing of the contents of a page. If you really want to, though, you can use absolute sizes pretty much anywhere, as a lot of issues with browsers have been worked out, but it's still a lot easier to stick with the convention and use relative sizes in most cases.

The rest of this chapter takes a closer look at some of the most common units, their uses, and some of their caveats.

3.3 Pixels (and their less-used cousin, the point)

The pixel (px) and point (pt) units in CSS are absolute measurements that are defined as 1/96 of an inch for a pixel, and 1/72 of an inch for a point. From this point on we're going to ignore the point because no one really uses it for web sizing (it's a holdover from the world of print design), and because it work justs like pixel (just with a different base size). Pixels, on the other hand, are most definitely useful (Box 3.1).

Box 3.1. Style Note: Anti-pixel Fundamentalists

As you keep progressing in your understanding of web development, you will inevitably come across people who are on an anti-pixel crusade. They believe that you should never use absolute sizes and do everything relatively. The truth is that there is a time and a place for everything, even absolute units like the pixel.

Some measurements just *feel* better or make more sense in pixels (hooray for subjectivity!), such as margins and padding (Chapter 4). Sure, you could define those using some of the relative measures we'll learn about below, and define all padding and margins based on the size of the size around it, but sometimes it is nice to know that whatever zoom level a user goes to there will be exactly 40 pixels of padding or margin around an element.

The real determining factor is how what you are designing is going to be used. If that pixel unit makes your job easier, and it is used in a place where it will never cause the layout to break, then go ahead and use it. Just know that if you get complaints from users that the site is breaking you'll probably have to redo the sizing to be relative!

Using an absolutely defined unit is great if you want to define the size of an element in a way that isn't dependent on the size of the browser or screen resolution, or anything else on the page, but it can lead to situations where the size of elements is completely inappropriate for a user's device. This type of unit isn't an inherently good or bad thing; you just need to be aware that anything sized using an absolute measurement is not going to be resized relative to anything else on the page—many sites have a mix of absolutely sized and relatively sized elements. The trick comes in knowing when to use the different units.

For instance, if you have banner ad images on your site, you are selling that advertising space based on the size of the element (one of the most common ads is the 728x90 leaderboard ad, as in Figure 3.8). In that situation, you want the ads to be defined with absolute dimensions and not

be relative (after all when you think about it, you are selling screen real estate, you want to make sure the size of the ad is what they paid for).

Figure 3.8: You've seen this sort of ad in a million places.

Images are also always sized by the browser, so that 1 pixel of the image equals one pixel on the browser. It's possible to have the browser resize images and use relative sizes, but the underlying default size of the image is 1 pixel to 1 pixel. This also is why you should never scale an image up from its default size in a browser—resizing should only be done to make an image smaller than it really is; otherwise, the browser has to spread pixels from the image over multiple pixels on the screen, and the image will look terrible (Figure 3.9).

Figure 3.9: Bad image resizing is bad.

So, that all still sounds reasonable right? Why can't we just use `px` measurements for everything if 96 pixels equals an inch? Well, not so fast. It turns out that a "screen inch" made up of 96 pixels isn't actually always the same as a real-world inch—for that measurement to be accurate, every screen's pixel density would have to be 96 pixels per inch.

Unfortunately, this is not the case. Modern smartphones and high-end displays have pixel densities of 400 pixels per inch and up, and the operating systems that run on them are often scalable, meaning that what you see on the screen can be resized in a way that isn't dependent on the physical number of pixels in the display—a bunch of smaller physical pixels are packaged together into a bigger "virtual" pixel. The point is, pixels at the software level no longer are directly related to physical pixels. As a result, the exact length of a 96px line is difficult to guarantee in a way that is universal for all users and screens.

One place where you will often see pixel sizes used online is in the definition of *font sizes*, which determine the size of text on the page. (This is also a place where you will find the most heated

"pixel vs. non-pixel" holy wars.) Part of the reason for the prevalence of absolute sizing in fonts is that it's a holdover from the days when pixel sizing was the only way to define font sizes on computers, and part of it is because there were people who were used to the conventions of print design where there were design requirements that said "this font should be exactly 24 points". When transitioning from print to screen, people familiar with absolute print sizing just brought their habits over to the web. So if a design in Photoshop had font that was 24pt, they'd make the design on the site 24px.

Absolute sizes were fine for the period when pretty much all screens had the same characteristics (and there were no other options), but over time screen sizes, resolution, and densities proliferated, and the inflexibility of absolute font sizes made relative sizing the preferred method.

There is another caveat that should make you not want to use pixels for fonts: If you set font sizes using pixels and then want to change some sizes later (either for a mobile-specific view or just because you didn't like the way it looked), you will have to go through and change every place where you defined a font size. If you had used relative values for those fonts, you could make changes in only one place, and everything else would inherit the new styling while still displaying in proportion. Making all text on a page bigger, or smaller, could be as easy as changing a single value.

So, with all those warnings about how you shouldn't use pixel sizes for fonts... let's use pixel sizes for fonts, and for element sizing, in a little quick pixel exercise! We'll make the `.bio-box` es have a width of `200px`, and set the font size of the `h2` element to be `30px`, as shown in Listing 3.5. (We'll undo this font-sizing in Section cha:values_rem.)

Listing 3.5: *Styling elements with pixel values.*

index.html

```
/* GLOBAL STYLES */
.
.
.
h2 {
  font-size: 30px;
}

/* BIO STYLES */
.
.
.
.bio-box {
  border: 1px solid black;
  width: 200px;
}
.
.
.
```

After saving and refreshing, your page should look like Figure 3.10.

Figure 3.10: Hey look, resized elements. Woo.

3.3.1 Exercises

1. You can (and should) use pixels to set the width of borders. Try changing the width of the `.bio-box` borders to `10px` .

3.4 Percentages

We've already sized elements using percentages earlier in the tutorial (Section 2.3), and from that you might be able to guess how useful this measurement can be when you are trying to use relative sizes to force an element to fill a space. There are a few catches, though:

- Percentage sizing is based on the parent container that an element is wrapped by—it isn't determined by the size of the browser, or the page as a whole.
- Percentage heights are a little weird because they require a set height on the parent element—they can't just assume a height the way that they assume a width.

Let's take a look at how percentages work, and you'll see what we are talking about. Add a new `div` that wraps the four `.bio-box` es, and set the class of that div to `bio-wrapper` (Listing 3.6).

Listing 3.6: *Adding a wrapper around the bio-boxes and giving it a class.*

index.html

```
.
.
.
<div class="bio-wrapper">
  <div class="bio-box">
    <h3>Michael Hartl</h3>
    <a href="http://twitter.com/mhartl" class="social-link">
      here
    </a>
    <div class="bio-copy">
      <p>
        Known for his dazzling charm, rapier wit, and unrivaled humility,
        Michael is the creator of the
        <a href="http://railstutorial.org/">Ruby on Rails
        Tutorial</a> and principal author of the
        <a href="https://learnenough.com/">
        Learn Enough to Be Dangerous</a> introductory sequence. Michael
        is also notorious as the founder of
        <a href="http://tauday.com/">Tau Day</a> and author of
        <a href="http://tauday.com/tau-manifesto"><em>The Tau
        Manifesto</em></a>, but rumors that he's secretly a supervillain
        are slightly exaggerated.
      </p>
    </div>
  </div>
  <div class="bio-box">
    <h3>Lee Donahoe</h3>
    <a href="https://twitter.com/leedonahoe" class="social-link">
      here
    </a>
    <div class="bio-copy">
      <p>
        When he's not literally swimming with sharks or hunting powder stashes on
        his snowboard, you can find Lee in front of his computer designing
        interfaces, doing front-end development, or writing some of the interface
        -related Learn Enough tutorials.
      </p>
    </div>
  </div>
  <div class="bio-box">
    <h3>Nick Merwin</h3>
    <a href="https://twitter.com/nickmerwin" class="social-link">
      here
    </a>
    <div class="bio-copy">
      <p>
        You may have seen him shredding guitar live with Capital Cities on Jimmy
        Kimmel, Conan, or The Ellen Show, but rest assured Nick is a true nerd at
        heart. He's just as happy shredding well-spec'd lines of code from a tour
        bus as he is from his kitchen table.
      </p>
    </div>
  </div>
  <div class="bio-box">
    <h3>??</h3>
    <div class="bio-copy">
      <p>
        The Future
      </p>
    </div>
  </div>
</div>
.
.
.
```

This wrapper is going to be the parent container that determines the size of the children `.bio-boxes` for which we'll set a percentage width. Add a style declaration that sets the width of this new class to be `500px`, and also change the width of the `.bio-box` es from the rather squished `200px` in Listing 3.5 to `50%`, as seen in Listing 3.7.

Listing 3.7: *Changing the width of the parent element.*

index.html

```
.
.
.
.bio-wrapper {
  width: 500px
}
.bio-box {
  border: 1px solid black;
  width: 50%;
}
.
.
.
```

Still pretty squished, right (Figure 3.11)?

Figure 3.11: Still not a lot of room for those boxes.

Those boxes are small because they are each taking their width setting as 250px (50% of the 500px wide parent). In order to get them to stretch across the page, we need to make their parent stretch all the way across the page, too. One way to do that is to remove the width from `.bio-wrapper` and then save and refresh. Your browser will assume that the percentage width for the `.bio-boxes` should be based on the width of the browser (Listing 3.8).

Listing 3.8: *Style declarations can be empty.*

index.html

```
.
.
.
.bio-wrapper {
}
.
.
.
```

Figure 3.12: The boxes are now taking up 50% of the width of the screen.

Now that the parent goes all the way across the screen, the child boxes definitely look bigger (Figure 3.12)!

Like we said above, percentage units work well for dimensions like width, but they behave a little strangely for height, and they don't work at all for thickness—meaning you can't use percentages for borders. In order for a percentage height to have an effect, the parent needs a set height (and even then things can get weird).

So, what if you wanted a box that was the height of the browser window? You'd think that setting a style to make it `height: 100%` would do the trick, but it won't work. To see what we mean, add the code from Listing 3.9 at the top of your page above the `h1` .

Listing 3.9: *Adding a test element for a quick and dirty example.*

index.html

```
.
.
```

```
.
<div style="border:1px solid #000;width: 50%;height:100%;">I'm a percent
test</div>

<h1>I'm an h1</h1>
.
.
.
```

That will give you a box that is half the width of the page, but surprisingly only the height of the content that is inside of the `div` (Figure 3.13).

Figure 3.13: Not quite what you'd expect for `height: 100%`.

Let's see what happens if instead we add a style that sets the height of the `body` element of the page, as shown in Listing 3.10. When you save and refresh, you'll find that your "percent test" box has grown to be really tall (Figure 3.14).

Listing 3.10: *Adding an absolute height to the parent.*

index.html

```
.
.
.
/* GLOBAL STYLES */
body {
  height: 800px;
}
.
.
.
```

Figure 3.14: A more effective `height: 100%`.

What happened there? When you specify a height in percent, the parent container has to have some sort of defined height set in order for the child percent height to be applied. Otherwise the browser just makes the element the height needed to contain the content within. Because we added a height to the `body` tag, which naturally has to wrap all the content, it ended up working more like a suggested height.

Even though we made the percent height work, at times it will act differently from how you expect, as it is dependent on the end height of the parent container. If the content in the parent causes the element to expand to an unexpected size, you might end up with the percent height child display on screen as far larger than you wanted. Or, if you inadvertently add a style that removes the set height from the parent, the child element with a percentage height would suddenly return to being the height only of the content it contains (like in Figure 3.13). Don't worry if this sounds confusing—just bear in mind that percent height is one of those properties whose behavior is liable to be unpredictable.

We'll look at some other solutions for getting elements to take up a proportion of a space later in the tutorial, both when we cover ways to make elements the size of the browser with the `vh` and `vw` units in Section cha:values_view, and then arrange to take up the full height of arbitrarily sized parents using the *flexbox* method in Chapter 7.

In the meantime, you should delete the "percent test" `div` from Listing 3.9, and make sure also to delete the style from Listing 3.10.

3.4.1 Percentage fonts

You *can* use percentages to set text sizes, but there is a consideration that you will have to keep in mind. If you use a percentage for a text size, the resulting size of the font is based not on the pixel dimensions of the container but rather on whatever `font-size` style that container has inherited. So the box itself could be `1000px` tall, but if it inherited a font size of `16px`, and you set the font size of a child element to 50%, you are going to get only an `8px`-tall font (50% of `16px`), not a `500px`-tall font like you might think.

The fact that percentage sizing uses different sources for sizing the height and width of boxes (based on the actual pixel dimensions of an element) versus sizing text (based on inherited font size) is why it is only infrequently used for text sizing. Most people find it easier to think of percentages as a way of determining the size of box-shaped things, and use other relative sizing methods for fonts.

If that all sounds confusing, don't worry—Section cha:values_em will explain in greater detail using the `em` unit, whose behavior is similar to percentage. However, unlike percentage measurement, `em`s are regularly used for sizing text and less used for sizing boxes. (*Warning*: This is a potential holy war statement to make.)

3.4.2 Exercises

1. Try setting a 10% margin on the `.bio-box`. This will push each box away from its neighbors by 10% of the width of the parent container, even the vertical margins (sharp-eyed readers might notice something weird with the vertical margins that we'll discuss in Section 4.6).

2. To see the way that percentage font sizes can be cumulative, set `.bio-box` font size to 150%, and then also set `.bio-copy` to 150%. The end result will be 150% of 150% of 16px (the page base font size), or in pixels...

3.5 em

The `em` unit is a relative size unit that is commonly used for sizing text (most would say it is the preferred method). The name comes from the approximate width of the letter *m*, but this usage is mainly historical; in CSS, one em represents a number of pixels equal to the current font size of any given element's parent container. If there is no font size that is inherited, then the default page font size is used.

For plain text (that is, not something like an `h1` header), the default size is `16px`, so the default size of an em is also `16px`. Fractions of an em then represent fractions of the full font size; for example, if the font size is `16px`, a unit of `0.5em` would be 50% of 16, or `8px`, and `2.25em` would be 225% of 16, or `36px`.

One of the things that makes ems useful, in comparison to something like pixel sizing, is that they automatically change value based on the font size that is inherited by the parent object that they are contained in. This means that if you used em sizing throughout your site, you can modify the entire site's text simply by changing a single base font size, and all the fonts in all the child containers will resize in correct proportion based on this new declared font size. If you used pixels for everything, you'd have to change every declared font size by hand.

As an example, let's say we set the font size of the `.bio-copy` to `0.5em`, as shown in Listing 3.11. Because the default base font size of the entire page is `16px`, the result in the browser is tiny `8px` text, as shown in Figure 3.15.

Listing 3.11: *Changing the bio copy font size.*

index.html

```
.
.
.
/* BIO STYLES*/
.
.
.
.bio-copy {
  font-size: 0.5em;
}
.
.
.
```

Figure 3.15: The bio copy text after shrinking down to `0.5em`.

Now let's see the effect of changing the font size of the parent element. In Listing 3.6 we added a `.bio-wrapper` div to wrap the bio boxes, so we can redefine the base font size by adding a CSS

rule as shown in Listing 3.12. The new rule in Listing 3.12 changes the font size from the default `16px` to `24px`, so the `.bio-box` font size of `0.5em` is now 50% of 24, or `12px`.

Listing 3.12: *Setting a new base font size.*

index.html

```
/* BIO STYLES */
.
.
.
.bio-wrapper {
  font-size: 24px;
}
.
.
.
```

To determine what the actual font size should be, the browser crawls up the parent/child tree until it finds a parent with a font size set with an absolute value, and then it calculates back down the tree to set the font sizes. As noted above, if there is no such absolute value the page default is `16px`, but by changing the parent div to `24px` we've managed to change the default for all child elements.

As a result, the font size of `0.5em` is no longer 50% of 16, but rather is 50% of 24, for a total of `12px`. The font size of the bio boxes automatically increases from the `8px` shown in Figure 3.15 to `12px`, as shown in Figure 3.16.

Figure 3.16: The bio copy in the `.bio-box` es has automatically increased.

One important property of `em` units is that they are *cumulative*: if an element that has its font size set to `0.5em` appears inside an element whose font size is also `0.5em`, then the resulting

font size for that bottom child element is 0.5 × 0.5 = `0.25em`. In real numbers, if the base size is `24px`, this means that the most deeply nested element is 25% of 24, or `6px`. This cumulative effect can be helpful, or it can cause unintentional display errors—you just have to be careful.

Schematically, our current page consists of nested divs, with `.bio-copy` inside `.bio-box` inside `.bio-wrapper`. We already changed the font size of `.bio-copy` to `0.5em` (Listing 3.11); if we change the `.bio-box` font size to `0.5em` as well, the result will be 50% of 50% of 24, or `6px`.

Listing 3.13: *Adding a relative font size to the* `.bio-box`.

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  font-size: 0.5em;
  width: 50%;
}
.bio-copy {
  font-size: 0.5em;
}
.
.
.
```

Look at the resulting teeny tiny font (Figure 3.17)!

Figure 3.17: Now the font size is unreadably tiny!

As outlined above, this happened because starting from the `.bio-copy`, the browser goes up one level to the parent and sees that there is a font size set, so it says, "OK, the font should be half the size of this parent size, but the `.bio-box` size is relative too, so let's go up again until we find an absolute font size." Going up one more level, the browser finds that `.bio-wrapper` declaration sets the font to `24px`, so now it can work back down and set the `.bio-box` to `12px`, and then the content inside the `.bio-copy` to `6px`.

This works the other way too. If we set both `.bio-box` and `.bio-copy a` font sizes to `1.5em` we'll end up with giant font that is the equivalent of `54px` font (24 × 1.5 × 1.5 = `54px`), as shown in Figure 3.18.

Figure 3.18: Now the font is just ridiculously huge.

Now that we've seen some crazy values, let's change the bio box to use a more sensible font size of `1em` (Listing 3.14).

Listing 3.14: *A sensible font size for the bio box.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  font-size: 1em;
  width: 50%;
}
.bio-copy {
  font-size: 1em;
}
.
```

So far we've used ems only for fonts, but the `em` unit *can* be used for things like margin, padding, and width (Chapter 4). In those cases, you have to remember that the size of an `em` is based on the local font size, so if you set a width for an object in `em` it will size the object based on the font size inside that element. For example, if the calculated font size in an element ends up being `16px`, and you set the padding to `1.5em`, then the padding will end up getting set to $1.5 \times 16 =$ `24px`.

Sound confusing? Figure 3.19 shows a quick diagram to explain.

Figure 3.19: Dimensions inside the box get calculated based on the font size.

The argument for doing your sizing using ems is that all your elements, and their attributes like padding or margin, will be sized relative to the size of text. But just because we want to increase the size of fonts on a page doesn't necessarily mean that we also want to change something like margin or padding. Sometimes you want the boxes that hold the content to stay the same and only the stuff inside to change, so styling every dimension of a container based on the text it contains can be inconvenient. As a result, in this tutorial we'll use the `em` unit only for fonts.

Warning: As is often the case with such subjective judgments, this is holy-war territory (Figure 3.20).[3]

Figure 3.20: Well... that's just, like, your opinion, man.

3.5.1 Exercises

1. Give the `.bio-box` class a padding of `2.5em`, and see how at small sizes the padding seems reasonable.
2. Set the `.bio-box` font-size to `48px`, and see how now your boxes' padding is a significant percentage of the entire screen width.
3. Set the `.bio-box` padding to a pixel size of 20px to see how that can make the spacing sizing independent from the content sizing.

3.6 rem isn't just for dreaming

The cumulative effect of the `em` unit (Section cha:values_em) can at times make designing layouts difficult since it makes it harder to drop sections of a page into other sections and be confident that you aren't going to end up with some crazy cumulative sizing issue. (Recall the goal from Section 2.4 to make our markup as modular and Lego-like as possible.) In recent years browsers have implemented a new relative unit that allows for us to create modular sections that can be placed on the page without sizing uncertainties. One of the most useful of these units is the *root em*, or `rem`.

This `rem` unit works similarly to `em`, in that it is a percentage of an absolute font size, but instead of being cumulatively sized based on the whole parent/child tree, the `rem` unit *always*

refers back to the font size of the `html` tag—in other words, it always refers to the most basic font size for the whole page. (As noted in Section cha:values_em, this default size is `16px` .)

In effect, the `rem` unit works like a document-wide setting, so you can set the size of elements like boxes, or font sizes, and have them all tie back to a single value: the font size of the `html` element. If you want to make everything a little bigger, or smaller, you can change just this one font size and everything on the page adapts.

`rem` is especially useful in combination with `em` units in developing modules. The best practice is to set a font size for the module's wrapper using a `rem` unit, and then style the fonts inside using `em` units. Because `rem` values are absolute (in relation to the page font size), you don't need to worry that the cumulative nature of `em` is going to keep going up the parent tree and make everything in the box tiny or huge (Section cha:values_em). This kind of styling allows you to create modules that can be safely dropped into any part of a page, while keeping the advantages of using relative font sizes.

To see this in action, let's set the size of the `.bio-box` to `1rem` and then add in a new declaration to set the header `h3` s to be `1.5em` and the `.bio-copy` to be `1em` .

Listing 3.15 has the entire CSS block at this point—copy and paste if you aren't synced up. (Note that Listing 3.15 removes the `h2` rule from Listing 3.5.)

Listing 3.15: *The CSS section up to now, with new font sizes.*

index.html

```
.
.
.
<style>
  /* GLOBAL STYLES */
  a {
    color: #f00;
  }

  /* SOCIAL STYLES */
  .social-link {
    background: rgba(150, 150, 150, 0.5);
    color: blue;
  }

  /* BIO STYLES */
  .bio-wrapper {
    font-size: 24px;
  }
  .bio-box {
    border: 1px solid black;
    font-size: 1rem;
    width: 50%;
  }
  .bio-box h3 {
    font-size: 1.5em;
  }
  .bio-copy {
    font-size: 1em;
  }
  .bio-copy a {
    color: green;
```

```
  }
</style>
.
.
.
```

Now the whole page will be set to be the same size as the `html` default font size of `16px`, and the header in the bio box will always be one and half times that size even though the `.bio-wrapper` is set to a very large `24px`. Meanwhile, the bio copy will remain at the default size for the page. If we decide that all the copy on your site should be bigger, we can easily reset the default size with a rule like

```
html {
  font-size: 18px;
}
```

With the change in Listing 3.15, all the copy in the boxes will resize but stay in proportion without any cumulative effects (Figure 3.21). All of those benefits become important the moment that you start needing to design a site that looks good on different devices, such as a desktop computer and a mobile phone. We'll discuss this important issue further in Chapter 9. (If you added the 18px font size styling to the `html` element, go ahead and delete it now.)

Figure 3.21: 16px base font size on the left, 18px on the right, with everything scaled proportionally.

3.6.1 Exercises

1. Copy the entire first `.bio-box` and paste it inside of the `h1`. You should see that `rem` sizing allowed for the whole section to be modular and retain the set styling.

2. In the CSS, change the font-size for `.bio-box` from `1rem` to `1em` and notice the effect.

3.7 vh, vw: The new kids on the block

Speaking of mobile-friendly units, we arrive now at two newer dimensional units that are also incredibly useful for responsive (mobile) layouts: the viewport height, `vh`, and viewport width, `vw`. These units allow us to size elements on the page based on the actual size of the browser window or mobile device screen. Each `vh` or `vw` is 1 percent of the corresponding screen dimension, so `3vh` would equal 3% of the height of the screen and `100vw` would be 100% of the width.

Neither `vh` nor `vw` is affected by parent elements, and neither has any weird cumulative inheritance issues—everything is determined by the size of the browser window or device screen. Up until recently these units weren't widely supported, but as long as a good percentage of your users aren't using really old browsers you can safely use `vh` and `vw` to do some fun things like design sections that fill the browser window no matter what the size of that window is.

We'll apply the viewport units as part of adding a *hero section* to our site, a recent design trend that involves having an attention-grabbing area at the top of the page containing a dramatic image, a call to action, etc. We'll start by wrapping the top section of our test page in a new `div` with two classes, `.full-hero` and `.hero-home` (Listing 3.16). (We'll use `.full-hero` starting in Section cha:box_sidebyside-float and `.hero-home` starting in Chapter 6.)

Listing 3.16: *Adding a wrapper around the content, and giving it class names.*

index.html

```
.
.
.
<div class="full-hero hero-home">
  <h1>I'm an h1</h1>
  <ul>
    <li>
      <a href="http://example.com/" class="social-link">Link</a>
    </li>
    <li>
      <a href="http://example.com/" class="social-link">Link</a>
    </li>
    <li>
      <a href="http://example.com/" class="social-link">Link</a>
    </li>
  </ul>
</div>
.
.
.
```

Note that Listing 3.16 also includes the results of the exercise in Section 3.1.3 that added the `.social-link` class to the link inside each `li`.

With the classes defined in Listing 3.16, we're in a position to start giving the hero section some styles. We'll start by adding a background color and a height equal to 50% of the viewport using

`50vh` , as shown in Listing 3.17.

Listing 3.17: *Adding a height based on browser size.*

```
index.html

/* GLOBAL STYLES */
.
.
.
/* HERO STYLES */
.full-hero {
  background-color: #c7dbfc;
  height: 50vh;
}
.
.
.
```

Note that the box isn't all the way to the top, and that there's extra space on the top, right, and left (Figure 3.22).

Figure 3.22: The box is now 100% of the page width and 50% of the height.

The extra space in Figure 3.22 is due to the browser's default *margin* on the `html` and `body` tags, a kind of spacing that we mentioned briefly in *Learn Enough HTML to Be Dangerous* and will cover more in Chapter 4. There is also an issue with the spacing around the `h1` , whose separate default margin is bleeding through the boundaries of the `.full-hero` parent due to something called *margin collapsing* (to be further explained in Section 4.2.1).

The solution is to *reset* the default styles using CSS. We'll implement a full CSS reset in Section 5.5, but for now let's apply a quick fix by adding the styles in Listing 3.18.

Listing 3.18: *Resetting the default margin and padding.*

`index.html`

```
/* GLOBAL STYLES */
.
.
.
html, body {
  margin: 0;
  padding: 0;
}
h1 {
  margin-top: 0;
}
.
.
.
```

Now the hero area takes up the whole top section of the site (Figure 3.23)!

Figure 3.23: Everything looks a lot nicer without the margins around the page.

If you've ever been to a site that has a big image that fills the top of the page, this is likely how they did it.

As with all the other values, viewport dimensions work for fonts too. For example, we can arrange for the `h1` to have a font size equal to 7% of the width of the browser window using `7vw` (Listing 3.19).

Listing 3.19: *Defining a font size relative to the browser width.*

`index.html`

```
.
.
.
/* GLOBAL STYLES */
.
.
.
h1 {
  font-size: 7vw;
  margin-top: 0;
}
.
.
.
```

One possible result appears in Figure 3.24, but if you resize the browser you'll notice that the font size changes.

Figure 3.24: Nice to see no margin around the edges and some big type.

This sort of dynamically resizing font may seem simple, but it was practically impossible for most of the history of the Web, so in reality it's pretty mind-blowing (Figure 3.25).

Figure 3.25: Fonts that resize based on viewport size = MIND. BLOWN.

As useful as the viewport dimensions are, if we used them for everything our site would look bad on either mobile or desktop (depending on which platform you were using when you designed the page). Desktop windows are just so much bigger than mobile that elements designed to fit a mobile screen would look giant, and elements designed for desktop would look impossibly tiny on mobile.

The solution is to use a *media query*, which allows us to set different styles for elements based on the size of the user's browser window. We'll cover this important technique in Chapter 9.

3.7.1 Exercises

1. We used the `vw` unit only on a font, but of course it works for element widths too. Using the `vw` unit, set the `.full-hero` class to take up 75% of the viewport width.

2. Guess what? We're going to blow your mind again... you can use `vw` to set a height, and `vh` to set a width! This is actually really useful when you want to make something responsively sized, but square in shape (so you need the height and width to be the same). To try it out, set both the height and the width of `.full-hero` to `50vw`.

3.8 Pleasing fonts

We'll end our discussion of CSS sizing with some notes on choosing good styles for text (e.g., font size).

When choosing font sizes, the goal should always be to have readable text on the page. Nicely laid-out copy should be the equivalent of somewhere between 14px and 18px in height.

The default size of 16px is right in the middle of this 14–18px range, but having a base unit of `1em` equaling `16px` can make the math a little difficult if you start using fractional sizes like `1.33em`. If for some reason you care about exact pixel sizes, there is a hack that you can use to combine em with pixel-precision—set a `font-size` on the `body` of `62.5%`, which makes `1em` elsewhere on the page equal `10px` (62.5% of 16). In web design circles, this is sometimes known as the "62.5% trick".

Although you are welcome to use the 62.5% trick, we think it's better to stop thinking in terms of absolute pixels when it comes to fonts—there's just no reason that you need to be exact on the sizing since there can be differences between browsers when displaying the same element. Also, even though the 62.5% trick gives you exact em sizing, you are also setting the default size of everything to a really small font size—for most people, `10px` is basically unreadable.

To sum up our recommendations: use relative units for text, don't use the 62.5% trick, and simply pick random numbers that set the font sizes somewhat close to what your design calls for (seriously).

What matters ultimately is how the end product looks and the relationship between different elements. Trying to achieve pixel-perfection is simply not a reasonable goal on the Web, and pixel sizing is really just an artifact from the bad old days of doing designs in Photoshop where elements heights and widths were set in pixels.

Just embrace the imprecision (Figure 3.26).

Figure 3.26: Just make sure that your imprecision is at least consistently around the target.

3.8.1 Exercises

1. Let's try the 62.5% trick. Set the `body` of your test page to have a font-size of `62.5%`. Notice that most text got smaller, with the exception of the `.bio-boxes` which are still being sized off of the `html` font size thanks to the `rem` unit.

2. Change the font size of the `h1` to `20px`, and then after seeing what it looks like change it to `2em`. There should be no change.

3. Delete the font size that is on the `.bio-wrapper` class, and then set the font size of the `.bio-box` to `1em` so that you can see what the new default size looks like.

Chapter 4
The box model

In Chapter 3, we saw how to set sizing parameters; in this chapter, we'll apply such sizing to one of the most important concepts in web design: the *box model*.

You may recall from *Learn Enough HTML to Be Dangerous* that when rendering HTML, the browser considers a page to be a collection of different boxes that contain content. Along with height and width, boxes can be styled to have *borders* (a line around the box), *margins* (the distance away from other boxes), and *padding* (empty space inside the box separating content from the border).

The CSS *box model* is the name for all the rules that determine how height, width, margin, padding, and borders are applied to elements (Figure 4.1). Elements of the box model can be quite confusing, with weird interactions between elements, counterintuitive applications of styles, and ways of writing style values that can look strange at first glance. In this chapter, we'll take a tour through these different idiosyncrasies, learn a couple of methods for getting boxes to sit next to each other, and lay the necessary foundation for applying the box model to our full website starting in Chapter 5.

Figure 4.1: The default HTML box model.

4.1 Inline vs. block

We'll begin our introduction to the box model by discussing the different effects of spacing and borders on *inline* vs. *block* elements. These two types of elements, which we discussed in *Learn Enough HTML to Be Dangerous*, behave differently in the context of the box model, so it's important to clarify the differences at the beginning.

Elements that are considered *inline elements*, like `span` or `a`, are only allowed to have margins and padding applied to the left and right (not top or bottom), and they won't accept a width or height set by CSS. None of these restrictions apply to block elements.

Confusingly, some styles can cause an inline element to switch to be a block element. You already saw an example when we *floated* the linked cover image in *Learn Enough HTML to Be Dangerous*. Floated elements become block elements and can suddenly have top and bottom margins or padding, plus active dimensions like height and width that were previously ignored. Changing an element's position on the page can also switch it from inline to block (as discussed further in Section 5.8).

You don't have to rely on these quirks to change elements from inline to block though—you can also directly force an element to change using CSS. There are actually a bunch of display property values that affect the way elements are drawn, with more being added all the time. In this tutorial, though, we are only going to consider five of the most important ones. Let's take a look!

4.1.1 display: none

The `display: none` style prevents the element from displaying on the page. For example, try updating the `.social-link` class rules to include `display:none` , as shown in Listing 4.1.

Listing 4.1: *Removing elements from the page.*

index.html

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 0.5);
  color: blue;
  display: none;
}
.
.
.
```

When you save and refresh, you'll see that all those social links are now gone (Figure 4.2). This style is commonly used for hiding elements in interactive websites, especially when combined with JavaScript (*Learn Enough JavaScript to Be Dangerous*). (Setting `display: none` here is just for demo purposes, so you should undo Listing 4.1 before proceeding.)

Figure 4.2: Hiding the display of social links with `display: none` .

To restore the display of an element that has been hidden, all you need to do is set the `display` property to anything other than `none` , such as `initial` or `block` . (In *Learn Enough JavaScript to Be Dangerous*, we'll use this technique to make hidden elements appear with the click of the mouse.)

4.1.2 display: block

`display: block` forces an element to be a block element regardless of what it was before. If you don't set dimensions after changing an element to `display: block`, it will behave like any normal block element by taking up the entire width of its parent element.

As mentioned briefly above, inline elements (such as links and spans) can't have a width or height, but once you change the display property the dimensional styles get applied. To see how this works, let's first add a height to `.social-link` (Listing 4.2).

Listing 4.2: *Adding dimensions to an inline element won't have an effect.*

index.html

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 0.5);
  color: blue;
  height: 36px;
}
.
.
.
```

When you save and refresh, you'll notice nothing changed—that's because the `.social-link` s are inline elements. Now add in the magical `display: block` (Listing 4.3) and save.

Listing 4.3: *Changing the display property allows the dimensional styles to apply.*

index.html

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 0.5);
  color: blue;
  display: block;
  height: 36px;
}
.
.
.
```

Refresh your browser and you'll see that your social links are now 36px-tall block elements that stretch all the way across their parent elements (Figure 4.3).

Figure 4.3: Look at those beautiful big gray squares.

4.1.3 display: inline

`display: inline` turns a block element into an inline element (essentially the opposite of the `display: block` property). Any styles that don't apply to inline elements (such as width and height, top margins, or padding) will no longer be applied. In addition, the element will no longer be on its own line, but rather will flow with text like any other inline element.

4.1.4 display: inline-block

The `inline-block` property, which is a hybrid between inline and block, is a useful display setting, as it allows styling that normally works only on block elements—such as width and height, top margins, and padding—to be applied to a particular element. At the same time, it also lets the element as a whole act like an inline element. This means that text will still flow around it, and it will only take up as much horizontal space as it needs to contain the content (as opposed to the way that block elements stretch all the way across the page unless you give them a set width).

To see how this works, set the `.social-link` s to display as inline-block on your index page (Listing 4.4).

Listing 4.4: *Setting the display of social links to* `inline-block` .

index.html

```
.
.
.
```

```
.social-link {
  background: rgba(150, 150, 150, 0.5);
  color: blue;
  display: inline-block;
  height: 36px;
}
.
.
.
```

When you save and refresh, you'll see that the links have the height style applied, but they are only as wide as the content (Figure 4.4)

Figure 4.4: The links are now a combination of inline and block, stretching only as wide as the content.

Eventually, in Section 10.1.1, we are going to add in icons for the different social media websites, and we'll want these links to all have the same dimensions regardless of the content that is inside. To make sure that they are all exactly the same size, let's also add a width property to the social links (Listing 4.5).

Listing 4.5: *The* `inline-block` *display lets you add a width to an inline element.*

`index.html`

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 0.5);
  color: blue;
  display: inline-block;
  height: 36px;
  width: 36px;
}
.
```

Your social links will now be nice little gray squares like in Figure 4.5.

Figure 4.5: The links now have the same width and height.

So, where do sites use this CSS style? The `inline-block` declaration is especially helpful when making site navigation, or when styling a group of elements so that they are side by side. We'll discuss this aspect of `inline-block` further in Section 4.5, and then again when we make page navigation in Section 5.6.2.

4.1.5 display: flex

`display: flex` is a powerful display property that forces all child elements to fill the entire parent element, and is highly customizable to allow for incredibly useful layout possibilities. The flex property is something that solves some of the most difficult long-running problems in page layouts.

We aren't going to play around with `display: flex` here because it really needs a whole chapter of its own to properly understand—a task we'll undertake in Chapter 7.

4.1.6 Exercises

1. Add a new class to your CSS called `show` after the `social-link` class and give it a display value of `block` . Set the `.social-link` display property to `none` again. When

you save and refresh the page, all the links should be gone. Now pick one of them and add `show` as a second class. What happens?

2. Set the `.social-link` display property to `inline-block` again. Now let's try changing a block element to an inline element. Use CSS to target the `li` s that are inside of the `.full-hero` and set their display property to `inline` .

4.2 Margins, padding, and borders

One of the most common places where developers interact with the box model is when adding *margins*, *padding*, and *borders* to elements on the page—the `margin` and `padding` properties control the space around or inside of elements, while the `border` property specifies the appearance of the boundary of the box. In this section, we'll take a first look at how these styles affect the box model (which includes some surprises), and then in Section 4.6 we'll look in detail at how margin, padding, and border styles are used in practice.

We'll begin by investigating padding and borders, which are different than margins in a key respect. In particular, if you specify the width of a block element, like a `div` or a `p` , and then apply a border or padding to it, the additional border or padding will go *outside* of the content. That means that you can end up with an element that is bigger than the dimensions that you specified. You'd think that if you said something should be 200px wide, it would always be 200px wide... but no, in the default state, CSS assumes that when you set a size for an element you are only talking about *the content part* of an element (Figure 4.6). This tends to generate a lot of confusion for people learning CSS, as they automatically assume that elements and their content are the same thing. Let's look at an example.

Suppose that you make a `div` and apply the following style:

```
width: 200px;
padding: 40px;
border: 10px solid #c00;
```

In this case, the entire element will end up being 300px wide on the page: 200px for the content, 40px each for the left and right padding, and 10px each for the left and right border ($200 + 40 \times 2 + 10 \times 2 =$ `300px`). This is the scenario illustrated in the original diagram for the box model (Figure 4.1), reproduced in Figure 4.6.

Figure 4.6: The default HTML box model again.

It is also possible to fix the total width of the content box, and force the border and padding to fit inside. The way to do this is with the `box-sizing` declaration. To see how this works, let's add some throwaway elements and styles to the page (which you can delete after seeing how this works).

We'll start with the HTML shown in Listing 4.6, which you can paste under the `h2` on the test page.

Listing 4.6: *Adding some test elements to the page to demonstrate box model properties.*

index.html

```
.
.
.
<h2>I'm an h2</h2>

<div class="test-box">
  200px wide
</div>
<div class="test-box test-box-nosizing">
  200px wide + border + padding = 300px
</div>
<div class="test-box test-box-nosizing test-box-sizing">
  200px wide + border + padding + box-sizing: border-box = 200px
</div>
.
.
.
```

Then add the styles in Listing 4.7 to the bottom of your style block (we'll be deleting these too).

Listing 4.7: *Adding classes and styles for the test elements.*

index.html

```
<style>
.
.
.
  .test-box {
    background: #9db6dd;
    width: 200px;
  }
  .test-box-nosizing {
    border: 10px solid #000;
    padding: 40px;
  }
  .test-box-sizing {
    box-sizing: border-box;
  }
</style>
.
.
.
```

When you save your work and refresh the browser, you'll see an assortment of boxes of different widths (Figure 4.7). Note how the `.test-box-sizing` class forces the div to be 200px wide in total. The `border-box` property caused the browser to draw the borders and padding *inside* of the defined width.

Figure 4.7: Different results even though all elements are set to 200px wide.

4.2.1 Margin weirdness

So, we've talked about how the box model can behave unexpectedly when it comes to borders and padding, but what about margins? Well, you would expect that, when two elements that both have margins are on the page next to each other, their margins would always apply. For example,

if two elements both have `20px` of margin, you might expect that the elements would always end up being `20 + 20 = 40px` apart—but that isn't necessarily how it works.

Figure 4.8: How the box model deals with margins between block and non-block elements.

We'll start by creating a situation where margins *do* behave in the intuitively expected manner. We'll do this by changing the test boxes introduced in Listing 4.6 to take the form shown in Listing 4.8.

Listing 4.8: *Changing our test boxes to show expected margin behavior.*

index.html

```
.
.
.
  .test-box {
    background: #9db6dd;
    display: inline-block;
    margin: 50px;
    width: 200px;
  }
  .test-box-nosizing {
    border: 10px solid #000;
    padding: 40px;
  }
  .test-box-sizing {
    box-sizing: border-box;
    display: block;
    width: auto;
  }
</style>
.
.
.
```

When you save and refresh the browser, you'll see the boxes from Figure 4.7 all separated from each other by 50px (Figure 4.9).

Figure 4.9: Trust us, they are all separated by the same distance.

Now we'll remove the `display` properties, so that each of the `div` s introduced in Listing 4.6 is reduced to its default (block) styling, while also removing the width styles, as shown in Listing 4.9.

Listing 4.9: *Collapsing the margins.*

index.html

```
.
.
.
.test-box {
  background: #9db6dd;
  margin: 50px;
}
.test-box-nosizing {
  border: 10px solid #000;
  padding: 40px;
}
.test-box-sizing {
  box-sizing: border-box;
}
.
.
.
```

The result is to magically collapse the margins: now all the boxes are vertically separated by only 50px of margin, as shown in Figure 4.10.

Figure 4.10: Less space now that the margins have collapsed.

The reason that the first example worked as expected is that the first two elements weren't block elements, so the browser fully respected their set margins. Once they became block elements, though, the browser allowed for only one of the margins to apply.

The reason this exists goes all the way back to the bad old early days of HTML (Section 1.2.2), when most websites used browser defaults for all elements (because there was no CSS). Some block elements (like paragraph `p` s) have default top and bottom margins to keep text away from other elements to enhance readability, and if there were no margin collapsing then whenever you put two of these elements next to each other there would be too much space between them. So at some point early on it was decided that when two block elements with margins follow each other, one of the top or bottom margins is canceled out.

In the next few sections, starting with Section cha:box_sidebyside-float, we'll be looking at how to put boxes side by side. If you are going to do the exercises, save the test blocks and styles; otherwise, you can delete the HTML and CSS styles from this section.

4.2.2 Exercises

1. Let's see if the width of the elements matters to margin collapsing. Change the size of the first two block elements by adding a style to set the width of `.test-box` to `200px` , and set the width for `.test-box-sizing` to `auto` to unset the previously set width value.

2. How does the browser determine which margin value to use? Try adding a style that sets a top margin of `100px` on the `.test-box-sizing` class.

4.3 Floats

Now that we've seen some of the things to watch out for in the box model, let's start using it to style our sample site. One thing you'll often need when you are designing a site is for different elements to sit next to each other on the page, and new developers often run into problems with how the box model affects their attempts to accomplish this. Perhaps unsurprisingly, there are a bunch of different ways to do this using CSS, and all have different positives and negatives. No single technique can be used across an entire site, so let's get started by learning about floating elements.

In *Learn Enough HTML to Be Dangerous*, we used a property value called a `float` to move an image to the left side of a block of text. The idea is that when you set an element to float to the left or right (there is no `float: center`), all the inline content around it will flow around the floated element like water. Floated elements will always sit next to other floated elements on the same line, as long as there is horizontal room. If the elements are too wide, they will drop down to the next line.

Let's see this in action. We'll add `float: left` to the `.bio-box` class, and will also give the boxes some padding and a new (narrower) width. Listing 4.10 has the new styles, as well as all the other styles for the test page at this point in case you'd like to sync up again. (Note that Listing 4.10 has also removed the HTML and CSS from Section 4.2.)

Listing 4.10: *The entire* `index` *page as of this point.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      /* GLOBAL STYLES */
      h1 {
        font-size: 7vw;
        margin-top: 0;
      }
      a {
        color: #f00;
      }

      /* HERO STYLES */
      .full-hero {
        background-color: #c7dbfc;
        height: 50vh;
      }

      /* SOCIAL STYLES */
      .social-link {
        background: rgba(150, 150, 150, 0.5);
```

```
      color: blue;
      display: inline-block;
      height: 36px;
    }

    /* BIO STYLES */
    .bio-wrapper {
      font-size: 24px;
    }
    .bio-box {
      border: 1px solid black;
      float: left;
      font-size: 1rem;
      padding: 2%;
      width: 25%;
    }
    .bio-box h3 {
      font-size: 1.5em;
    }
    .bio-copy {
      font-size: 1em;
    }
    .bio-copy a {
      color: green;
    }
  </style>
</head>
<body>
  <div class="full-hero hero-home">
    <h1>I'm an h1</h1>
    <ul>
      <li>
        <a href="http://example.com/" class="social-link">Link</a>
      </li>
      <li>
        <a href="http://example.com/" class="social-link">Link</a>
      </li>
      <li>
        <a href="http://example.com/" class="social-link">Link</a>
      </li>
    </ul>
  </div>
  <h2>I'm an h2</h2>
  <div class="bio-wrapper">
    <div class="bio-box">
      <h3>Michael Hartl</h3>
      <a href="http://twitter.com/mhartl" class="social-link">here</a>
      <div class="bio-copy">
        <p>
          Known for his dazzling charm, rapier wit, and unrivaled humility,
          Michael is the creator of the
          <a href="http://railstutorial.org/">Ruby on Rails
          Tutorial</a> and principal author of the
          <a href="https://learnenough.com/">
          Learn Enough to Be Dangerous</a> introductory sequence. Michael
          is also notorious as the founder of
          <a href="http://tauday.com/">Tau Day</a> and author of
          <a href="http://tauday.com/tau-manifesto"><em>The Tau
          Manifesto</em></a>, but rumors that he's secretly a supervillain
          are slightly exaggerated.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <h3>Lee Donahoe</h3>
      <a href="https://twitter.com/leedonahoe" class="social-link">here</a>
      <div class="bio-copy">
        <p>
          When he's not literally swimming with sharks or hunting powder
          stashes on his snowboard, you can find Lee in front of his computer
          designing interfaces, doing front-end development, or writing some of
```

```
            the interface-related Learn Enough tutorials.
          </p>
        </div>
      </div>
      <div class="bio-box">
        <h3>Nick Merwin</h3>
        <a href="https://twitter.com/nickmerwin" class="social-link">here</a>
        <div class="bio-copy">
          <p>
            You may have seen him shredding guitar live with Capital Cities on
            Jimmy Kimmel, Conan, or The Ellen Show, but rest assured Nick is a
            true nerd at heart. He's just as happy shredding well-spec'd lines
            of code from a tour bus as he is from his kitchen table.
          </p>
        </div>
      </div>
      <div class="bio-box">
        <h3>??</h3>
        <p>
          The Future
        </p>
      </div>
    </div>
  </body>
</html>
```

After you save your work and refresh the browser, the result should appear as in Figure 4.11.

Figure 4.11: The boxes are next to each other, but they don't fit.

Now all the boxes are in a row, but why is the last one spilling over to the next line?

It's because of the box model sizing issue from Section 4.2! The left and right border, and the left and right padding, got added onto the width of each `div`, making each one `25% + 1px`

`border-left + 1px` `border-right + 2%` `padding-left + 2%` `padding-right` , for a total size of `29% + 2px` of the page for each. Multiplied by 4, this gives `116% + 8px` , which is greater than 100%.

Let's fix this by adding the `box-sizing: border-box` style to the `div` to force the borders and padding inside of the set-width `div` (Listing 4.11).

Listing 4.11: *Adding* `border-box` *to the bio boxes.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  padding: 2%;
  width: 25%;
}
.
.
.
```

Now when you save and refresh you'll have four boxes in a row that fill the page (Figure 4.12)!

Figure 4.12: Box-sizing saves the day, and now our floated boxes fit.

4.3.1 Clearing floats

So, why might a developer not want to always use floating to get elements to line up side-by-side?

For one, there are only two options, `float: left` and `float: right`, but no `float: center`. That's annoying, but manageable. The bigger problem is that the browser doesn't always know where to end the float. When you float elements, you are telling the browser that you'd like the element to show up on the page in the place it would naturally "float" to, but after that starting position you want the rest of the page content to flow around the floated element. This can disrupt the orderly box-like arrangement of elements, and create some odd-looking layouts.

To see what we mean, add the paragraphs from Listing 4.12 onto your test page below the closing `</div>` tag of the `.bio-wrapper`.

Listing 4.12: *Adding text to the page below the bios.*

index.html

```
.
.
.
  <div class="bio-wrapper">
    .
    .
    .
  </div>
  <p>
    Learn Enough to Be Dangerous is a leader in the movement to teach the
    world <em>technical sophistication</em>, which includes both "hard
    skills" like coding, command lines, and version control, and "soft
     skills" like guessing keyboard shortcuts, Googling error messages, and
    knowing when to just reboot the darn thing.
  </p>
  <p>
    We believe there are <strong>at least a billion people</strong> who can
    benefit from learning technical sophistication, probably more. To join
    our movement,
    <a href="https://learnenough.com/#email_list">sign up for our official
    email list</a> now.
  </p>
  <h3>Background</h3>
  <p>
    Learn Enough to Be Dangerous is an outgrowth of the
    <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a> and the
    <a href="http://www.softcover.io/">Softcover publishing platform</a>.
    This page is part of the sample site for
    <a href="https://learnenough.com/css-tutorial"><em>Learn Enough CSS and
    Layout to Be Dangerous</em></a>, which teaches the basics of
    <strong>C</strong>ascading <strong>S</strong>tyle
    <strong>S</strong>heets, the language that allows web pages to be styled.
    Other related tutorials can be found at
    <a href="https://learnenough.com/">learnenough.com</a>.
  </p>
.
.
.
```

When you save your work and refresh the page, you'll see that the floated elements have caused the text we just added to start under the rightmost float instead of starting on a new line (Figure 4.13).

Figure 4.13: This is not what the doctor ordered.

In an ideal world, those paragraphs should stretch across the entire page since they are block elements. One way to get things back to the expected result would be to use the CSS `clear` rule, which is used to let the browser know to end floats. In this case, we could add `clear: left` to the first paragraph.

You can try it by adding an inline style (Listing 4.13).

Listing 4.13: *A simple inline style to clear the float.*

index.html

```
.
.
.
<p style="clear: left;">
  Learn Enough to Be Dangerous is a leader in the movement to teach the
.
.
.
```

This will force the paragraph onto a new line below the floated elements, and it will prevent any other element below it on the page from being altered by the float (Figure 4.14).

Figure 4.14: This works, but you don't want to have to manage clearing things on potentially every element.

If the floating elements had been floated to the right using `float: right`, you would need to clear their float status with `clear: right`, or (if you just want to be extra careful) you can clear both types of floats using `clear: both`.

If you tried clearing the float with an inline style on your test page, you should remove the style from the `p` tag—just the mere sight of an inline style should make you feel queasy at this point (but they are handy at times for quickly testing styles).

Having to add a `clear` style directly onto any element (either inline or in the stylesheet) after floated elements is kind of a pain—especially on a dynamic site that might pull in snippets of code to build a page. You don't always know which elements will be following floats.

A better way to clear floats is apply a rule to clear everything inside a wrapper, such as the `.bio-wrapper` added in Listing 3.6. The idea is to arrange for the `.bio-wrapper` element, and everything in it, to act like a Lego block that can safely be moved around without needing to worry about uncleared floats messing up a layout.

There are two methods to clear floats inside a wrapper: the `overflow` method, and the `:after` "clearfix" method. We'll look at both of these methods here, and then cover a little more about the overflow property in Section 4.4. Then, we'll talk more about the `:after` declaration in Section 6.3.1.

To see the overflow method in action, add the following style from Listing 4.14 to `.bio-wrapper`.

Listing 4.14: *When overflow is set to hidden, floats are cleared.*

index.html

```
.
.
.
/* BIO STYLES */
.bio-wrapper {
  font-size: 24px;
  overflow: hidden;
}
.
.
.
```

When you save and refresh, the paragraph of text will be safely below the floated elements with no inline styles, and no use of the `clear` property (Figure 4.15).

Figure 4.15: Same result, but self-contained instead of needing inline styling.

The problem with this method is that if you also need to set a height or width on the element that has `overflow: hidden` set, the content inside can get cut off. This happens most often with poorly built *dropdown menus* in a site navigation with floats that are cleared using the overflow method but where the header also had a set height. For example, Figure 4.16 shows what Amazon.com's homepage dropdown menu would look like if they had mistakenly also set `overflow: hidden`.[1]

Figure 4.16: Changing the overflow on an element can hide parts that were supposed to stick out.

So, if you need to clear floats but also are worried about content being cut off because you absolutely, positively *have* to set a height on the wrapper element, you can use the `:after` method.

Let's see how it works. Remove `overflow: hidden` from the `.bio-wrapper` class, and add in the entire new declaration from Listing 4.15.

Listing 4.15: *The more complicated "clearfix" method.*

index.html

```
/* BIO STYLES */
.
.
.
/* BIO STYLES */
.bio-wrapper {
  font-size: 24px;
}
.bio-wrapper:after {
  visibility: hidden;
  display: block;
  font-size: 0;
  content: " ";
  clear: both;
  height: 0;
}
.
.
.
```

There's a lot of new stuff in there, but don't worry about it for now. We'll discuss `:after` in more detail in Section 6.3.1. The important thing for now is that `:after` creates a kind of imaginary element that comes at the end of the `bio-wrapper` —an imaginary element that we can add styles to! Setting `clear: both` on that element clears the floats and allows the content

below to appear as intended. If you save the changes and refresh your browser, the text will still be cleared below the floated elements as in Figure 4.14 and Figure 4.15.

4.3.2 Exercises

1. Let's try floating something to the other side to see how floats can change the order of content. Change the float property on `.bio-box` to float to the right. Notice which box is now on the left.

2. Change the clear property in the `:after` style to clear the right. You should see that the content in the wrapper no longer is being cleared. You need to make sure to either match floats to clears or use `clear: both` !

4.4 A little more about the overflow style

In Section cha:box_sidebyside-float, we used the overflow method to clear floats, but you may be wondering exactly why this method works... and also, what does `overflow` do in the first place?

The CSS `overflow` property tells the browser how to handle content inside a wrapper, if that wrapper has a set height or width. If the content in the wrapper doesn't fill the box, then `overflow` does nothing, but when there is more content than room to display that content, overflow comes into play. Because this property can be used to clear floats, and control how content is displayed, it is worth exploring in detail.

The `overflow` style can be set to `visible` , which shows everything; `hidden` , which cuts content off at the boundaries of the wrapper; or `scroll` , which adds scroll bars to let you scroll up and down or left and right to see all the available content.[2] You've seen this before if you've ever scrolled inside a box on a website without scrolling the entire page (Figure 4.17).

Figure 4.17: Scrolling within a box on a page without scrolling the page.

`overflow: hidden` works for clearing floats because it makes the browser want to keep content contained entirely within the wrapper. If there is no set dimension on the wrapper, the browser just expands the boundaries of the wrapper to reach the end of the floated elements, and then lets the elements that follow display on the page normally.

Figure 4.18 has diagrams of some of the different possible overflow situations when containing floated elements, and why you have to be careful when adding a height to an element with overflow set to `hidden`.

Figure 4.18: Some examples of overflow and containers.

To see what happens with the different settings in practice, let's start with `overflow: hidden`, and also give the `.bio-wrapper` a background color and a height (Listing 4.16).

Listing 4.16: *Overflow set to hidden with a height set on the container.*

index.html

```
.
.
.
/* BIO STYLES */
.bio-wrapper {
  background-color: #c0e0c3;
  font-size: 24px;
  height: 300px;
  overflow: hidden;
}
.
.
.
```

Figure 4.19: Adding a height and setting to `hidden` cuts off the content.

You can see in Figure 4.19 that any content that is taller than the wrapper gets cut off. Now let's try `overflow: visible` (Listing 4.17).

Listing 4.17: *Overflow set to visible with a height set on the container.*

index.html

```
.
.
.
.bio-wrapper {
  background-color: #c0e0c3;
  font-size: 24px;
  height: 300px;
  overflow: visible;
}
.
.
.
```

You can see the content extending out past the boundary of the `.bio-wrapper` (Figure 4.20).

Figure 4.20: The content extends out of the green box due to `overflow: visible`.

Finally let's try setting the value to `scroll` (Listing 4.18). An overflow set to scroll with a height set on the container will keep all the content inside the container, but you can scroll to see it.

Listing 4.18: *Setting the* `overflow` *to* `scroll`.

index.html

```
.bio-wrapper {
  background-color: #c0e0c3;
  font-size: 24px;
  height: 300px;
  overflow: scroll;
}
```

Now the content is cut off, but if you put your cursor in the green box and scroll up or down with your mouse or trackpad you'll be able to see the hidden content (Figure 4.21).

Figure 4.21: A scrollbar appears on the right just for the green box.

If you tried out these styles on your test page, set the `.bio-wrapper` back to the styles in Listing 4.19.

Listing 4.19: *Returning the* `.bio-wrapper` *to* `overflow: hidden`.

index.html

```
.
.
.
.bio-wrapper {
  font-size: 24px;
  overflow: hidden;
}
.
.
.
```

4.5 Inline block

The second way to get things next to other things is to set the elements to be `display: inline-block` (Section 4.1.4). This allows them to keep their block-style formatting (so they can have height and top/bottom margin and padding), while also letting us do things like control an element's position on a line of text by setting the style `text-align` on a wrapper to make everything align left, right, or center.

For this example, we are going to style the `li` s that contain the `.social-link` s inside the `.full-hero` at the top of the test page. Let's first give the unordered list tag `ul` a class called

`.social-list` (Listing 4.20).

Listing 4.20: *Adding a class name to the unordered list.*

index.html

```
.
.
.
<ul class="social-list">
  <li>
    <a href="http://example.com/"
       class="social-link">Link</a>
  </li>
  <li>
    <a href="http://example.com/"
       class="social-link">Link</a>
  </li>
  <li>
    <a href="http://example.com/"
       class="social-link">Link</a>
  </li>
</ul>
.
.
.
```

By default, the `li` tag produces block elements that each start on their own new line; list elements that are part of a unordered list also include a bullet point before each element's content.[3] Why would we want to use such a list to display a person's social links?

The answer is that we can unstyle the list (removing the bullet points, making it inline instead of block, etc.) and use it however we want! It has become common practice to use the `ul` tag to contain sets of links for things like navigation, menus, etc., because it's the logical HTML element for grouping lists of things, which as designers gives us a nice structure to work with.

So, first let's unstyle that list, and then add a second declaration that targets the `li` s inside of `.social-list` and switches them to `display: inline-block` (Listing 4.21). By the way, the right angle bracket `>` on Line 12 in Listing 4.21 is a more advanced selector called the *child selector*, discussed briefly in Box 4.1 and covered in more depth in Section 5.7.3.

Listing 4.21: *Unstyling the list and making the `li` s inline-block.*

index.html

```
1 .
2 .
3 .
4 /* SOCIAL LINKS */
5 .
6 .
7 .
8 .social-list {
9   list-style: none;
10  padding: 0;
11 }
12 .social-list > li {
13  display: inline-block;
14 }
15 .
```

```
.
.
```

Hey, look at those elements all in a row (Figure 4.22)!

Figure 4.22: Our `li`s are now in a row and have no bullet points.

Notice that there are little spaces between the elements in Figure 4.22. This is an annoyance that comes with using this technique, and it stems from the way that browsers treat `inline-block` elements as though they are just funky-shaped words in a sentence. There are ways to get rid of that space, but we aren't going to get into those weeds here; instead, we are just going to ignore them since in our case it is nice to have a bit of space between elements. (For the record, when you use a float to get elements next to each other, there is no space between them at all.)

Box 4.1. Intro to Advanced Selectors

If you thought things weren't going to get any more complicated with selectors... you were wrong. We've barely scratched the surface! We are going to cover them more later in the tutorial (Section 5.7.3), but for now we want to explain the child selector.

Take a look again at the style we declared in Listing 4.21:

```
.social-list > li {
  display: inline-block;
}
```

What this says is "select only `li`s that are direct children of the `.social-list` parent, and make them `inline-block`." Remember that one of the goals when you are styling pages is to style only the things that need styling, without accidentally styling elements that will later need to be unstyled (Section 2.4). When you use advanced selectors, you can better target your declaration.

For example, suppose we had a second nested unordered list in one of those `li` s, like this:

```
<ul class="social-list">
  <li>
    <a href="http://example.com/"
       class="social-link">Link</a>
  </li>
  <li>
    <a href="http://example.com/"
       class="social-link">Link</a>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
    </ul>
  </li>
  <li>
    <a href="http://example.com/"
       class="social-link">Link</a>
  </li>
</ul>
```

In this case, the `li` s in the nested list would remain as block elements. This is because they are children of a plain `ul` element, and the CSS rule targets only children of a `ul` with class `social-list` . If you like, you can try out that example on your test page (and then delete it when you're done).

Now let's center the links (Listing 4.22).

Listing 4.22: *Centering the inline-block elements.*

index.html

```
.
.
.
/* SOCIAL STYLES */
.
.
.
.social-list {
  list-style: none;
  padding: 0;
  text-align: center;
}
```

Figure 4.23: A simple `text-align: center` *puts the boxes right in the middle.*

You have no idea how hard what we did in Figure 4.23 used to be before this `inline-block` technique was available. In the bad ol' early days of the Web, when we had to use tables for everything, it used to be a giant pain to get things to work right, but now it is just as easy as we just saw. We can play around with aligning the links left or right, and everything is nicely contained inside the `ul` without any need to clear floats.

4.5.1 Exercises

1. Add a number to the end of the link text in the hero so that they now read as "Link1", "Link2", etc. Now, align the `.social-link` s to the right by changing the text alignment property for `.social-list` to `right` . Notice that, unlike when we floated elements to the right, the order doesn't change.

2. Test out the child selector by creating a new style declaration that changes only links that are the direct children of the `.bio-box` es to `#c68bf9` . You'll see that this declaration has a high specificity, and will override the color set in the `.social-link` class.

4.6 Margins for boxes

Now that we've gotten a handle on arranging our boxes, let's go and look in detail at margin, padding, and borders. These styles allow developers to control the spacing between boxes (with `margin` in this section), the spacing inside of boxes (with `padding` in

Section cha:box_padding), and the size and look of the edges of boxes (with `border` in Section cha:box_border).

We'll start with the simplest kind of margin declaration, which we'll add to the `.bio-box` es at the bottom of the page, as shown in Listing 4.23.

Listing 4.23: *Adding a margin declaration.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 20px;
  padding: 2%;
  width: 25%;
}
.
.
.
```

When you refresh the test page, you'll see that each one of the containers has moved away from everything else by `20px` in all directions (Figure 4.24).

Figure 4.24: The boxes got some extra space, but broke onto two lines again.

The next question that you might be asking is, "Why are the boxes on two lines again? I thought we took care of that with `box-sizing: border-box` (Listing 4.11)?"

The answer is that in the box model, margins always apply *outside* of an element. So even though we've set the `box-sizing` style to `border-box`, the four `div`s at the bottom are now taking up 100% *plus* 8 * 20px, which is wider than 100%.

So, how do we get everything to fit again? First, to make this easier we are going to switch to using the same units for each measurement (all percentages), and we are going to have to do a little bit of math.

Let's first set the margins to be percentages, as shown in Listing 4.24.

Listing 4.24: *Changing the margin from pixels to percent.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 3%;
  padding: 2%;
  width: 25%;
}
.
.
.
```

Based on the result (Figure 4.25), this looks like a reasonable amount of spacing.

Figure 4.25: Pretty good margin size.

Now let's do the math. If we have a left and right margin of 3%, that means we need to reduce the size of each of the containers by 3% + 3% = 6% to get everything to fit. Because the original width was 25%, this means the new width should be 25% – 6% = 19%, as shown in Listing 4.25.

Listing 4.25: *Changing the width to accommodate the margins.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 3%;
  padding: 2%;
  width: 19%;
}
.
.
.
```

It fits again (Figure 4.26)!

Figure 4.26: Everything fits again.

Now let's take a look at the actual style declaration to dig a little deeper into how it works. When we write `margin: 3%` , we are applying the margin in *all* directions around the box, which is the equivalent of applying these four styles all at once:

```
margin-top: 3%;
margin-right: 3%;
margin-bottom: 3%;
margin-left: 3%;
```

The simple `margin` declaration that we used is a shorthand version that combines all the directions of the margin onto one line, and `margin: 3%` is the equivalent of writing this:

```
margin: 3% 3% 3% 3%;
```

As you may recall from *Learn Enough HTML to Be Dangerous*, the order here is top, right, bottom, left, as illustrated in Figure 4.27.

Figure 4.27: Think of the four values as going clockwise from the top.

What if you really only want the margin on certain sides of the `div`? For example, to set the top margin to `40px` and the left margin to `30px`, we could use the more specific declarations shown in Listing 4.26.

Listing 4.26: *Changing the width to accommodate the margins.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin-top: 40px;
  margin-left: 30px;
  padding: 2%;
  width: 19%;
}
.
.
.
```

That will work just fine, but our code will start getting cluttered if we have to define every direction for every declaration that has a direction option. A better way to control where the

margin goes is to use the single `margin` attribute and leverage the shorthand from Figure 4.27. To get margins of 40, 30, 40, and 30 pixels going around the `div` (clockwise from the top), we could `style` the margin like this:

```
margin: 40px 30px 40px 30px;
```

But guess what? As Figure 4.27 shows, in addition to the shorthand `margin: 40px` (using a single number), it's possible to include only two values if your top and bottom values are the same, and your left and right values are also the same (but different than the top and bottom):

```
margin: 40px 30px;
```

This shorthand also works with just three values, like `margin: 20px 10px 40px` . This is missing the final value, which (as seen in Figure 4.27) is the left margin, which will be filled in automatically from its opposite across the box (in this case, `10px`).

For our test page, let's set only a top margin of `40px` , and a right and left margin of `1%` , and also increase the size of each container back to `23%` so that the entire row fills the available space (Listing 4.27).

Listing 4.27: *Adding the margin declaration.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 40px 1% 0;
  padding: 2%;
  width: 23%;
}
.
.
.
```

After saving your work and refreshing the browser, you'll see that the `div` with the link has now moved down `40px` from the content above and 1% from the sides of the browser and from the other containers (Figure 4.28).

Figure 4.28: Much nicer spacing.

Looks great!

If you noticed that the margin between the boxes is different from the ends, that's because it's being doubled (due to the same margin on the left and right). We'll tackle this sort of issue with an advanced selector solution later in Section 5.7.3.

4.6.1 An exception: "margin: auto;"

You might be getting used to this by now, but of course there is another oddity that you should be aware of: `margin: auto` .

If you have a block element, like a `div` , `p` , or a `ul` , that has a width set by a style, you can make the browser center that element horizontally within its parent container by setting the left and right margin to `auto` .[4]

To see `margin: auto` in action, let's change the styling (Listing 4.28) for the `.bio-wrapper` to give it a `max-width` and then set the margin to `auto` . `max-width` is a CSS style that lets an element adapt its width to fit a space (up to a specified value), and there's also a `min-width` that does the opposite. Both are helpful when designing sites that are intended to look good on both mobile and on desktop, since on the smaller screen you want content to fill the browser, but on a big screen that could look sloppy.

Listing 4.28: *Applying* `margin: auto` *.*

index.html

```
.
.
.
.bio-wrapper {
  font-size: 24px;
  margin: auto;
  max-width: 960px;
  overflow: hidden;
}

.
.
.
```

Save and refresh, and watch the box magically center (Figure 4.29).

Figure 4.29: It's boxes... in the middle of the page!

4.6.2 Yet another exception: negative margins

Guess what? You can also make margins *negative* for elements. This draws the element up and out of the normal place it occupies on the page and overlays it on content that normally it wouldn't be able to affect.

To see this in action, let's first add some images (using pictures from the lighthearted placekitten website) to the `.bio-box` es. Place an image above the `h3` s in each `.bio-box` , as shown in Listing 4.29.

Listing 4.29: *Adding in an image to one of the .bio-boxes.*

index.html

```
.
.
```

```
.
<div class="bio-box">
  <img src="http://placekitten.com/g/400/400">
  <h3>Michael Hartl</h3>
  .
  .
  .
</div>
<div class="bio-box">
  <img src="http://placekitten.com/g/400/400">
  <h3>Lee Donahoe</h3>
  .
  .
  .
</div><div class="bio-box">
  <img src="http://placekitten.com/g/400/400">
  <h3>Nick Merwin</h3>
  .
  .
  .
</div><div class="bio-box">
  <img src="http://placekitten.com/g/400/400">
  <h3>??</h3>
  .
  .
  .
</div>
.
.
.
```

And then add a little bit of CSS to resize the images (Listing 4.30).

Listing 4.30: *A style to control the size of the images that we added.*

index.html

```
.
.
.
/* BIO STYLES */
.
.
.
.bio-box h3 {
  .
  .
  .
}
.bio-box img {
  width: 100%;
}
.
.
.
```

With that, the images nicely fill up the space (Figure 4.30).[5]

Figure 4.30: Your `.bio-boxes` should now look a little fluffier.

Now we'll add a negative top margin to the `.bio-box h3` s (changing from `margin-top` to the three-value `margin` shorthand mentioned in Section 4.6), as well as some extra styling for the text (Listing 4.31).

Listing 4.31: *Negative margin moves an element out of its natural position.*

index.html

```
.
.
.
.bio-box h3 {
  color: #fff;
  font-size: 1.5em;
  margin: -40px 0 1em;
  text-align: center;
}
.
.
.
```

Our new styles have pulled the header text out of its normal place and instead have drawn it on top of the images (Figure 4.31).

Figure 4.31: Negative margins move elements outside their natural position on the page.

Negative margins might seem like an odd property to allow, but it is actually useful from time to time. Negative margins also allow us to extend some content up and out of a box and overlap that content into a space where it normally wouldn't be able to be positioned, all while maintaining its properties as a normal block-level element.

4.6.3 Exercises

1. Use what we've learned about the margin shorthand to add in some space below the bottom of the `.bio-wrapper` , while using the CSS shorthand to keep the margin set to zero on top, and auto for the left and right.

2. Try to center the `.social-link` s that are inside of the `.bio-box` es using a margin set to auto. Why doesn't this work even though the elements have set widths?

4.7 Padding... not just for chairs

As we saw at the beginning of Section 4.2, padding is similar to margins, except instead of pushing away things *outside* the element, padding pushes the content *inside* an element away from the edges of the element. This is ideal when you want to have a box containing text have a background color or a border but you don't want the text ending up smashed against the edge of the container.

Padding values are declared using the same syntax as for margins, including the shorthand from Figure 4.27. Let's try it out and remove only the top padding from our `.bio-box` es by changing the styling to match Listing 4.32.

Listing 4.32: *Padding value shorthand works just like the margin shorthand.*

index.html

```
.
.
.
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 40px 1% 0;
  padding: 0 2% 2%;
  width: 23%;
}
.
.
.
```

The result of Listing 4.32 appears in Figure 4.32. (We'll revert this style choice in Section cha:box_border.)

Figure 4.32: Hey look, no top padding.

Padding is one of the easier to understand CSS properties, as it doesn't have a whole lot of weird exceptions.

4.7.1 Exercises

1. Add a padding of `20px` to the links that are inside of the `.bio-copy` sections. You'll see how top and bottom padding can't be applied to an inline element like a default link element, but the left and right padding will still be added.

4.8 Fun with borders

You might have wondered about the `border: 1px solid black` style that has been on the `div` s this whole time. As you might have guessed, this style declaration is also a shorthand similar to margin and padding, but slightly different in that it is three totally different style declarations condensed into one (instead of just different directions as with margin and padding).[6] The most common usage is to apply a border to all sides of an element, like this:

```
border: 1px solid black;
```

This is actually a condensed version of the following rules:

```
border-width: 1px;
border-style: solid;
border-color: black;
```

All of those styles behave like the margin and padding in that they are directional shorthand that applies styling to the top, right, bottom, and left, like this:

```
border-width: 1px 1px 1px 1px;
border-style: solid solid solid solid;
border-color: black black black black;
```

Note that the `border-style` declaration isn't a number, but rather can take on the following values: `none` , `hidden` , `dotted` , `dashed` , `solid` , `double` , `groove` , `ridge` , `inset` , and `outset` .

You might ask, "Well, that is all well and good, but what if I don't want all sides of a border to be the same?" The shorthand doesn't seem to cover that. What would the most efficient way be to make a 1px border that is colored black on all sides, but is red on one side (let's say the bottom side)? One way to achieve this look would be to separately declare all the different sub-declarations of the shorthand, like this:

```
border-width: 1px;
border-style: solid;
border-color: black black red;
```

Or you could do it in a more condensed way and take advantage of the fact that rules that come after a similar declaration take precedence (Section 2.3), as in Listing 4.33.

Listing 4.33: *Styling a border to have different colors on different sides.*

index.html

```
.
.
```

```
.
.bio-box {
  border: 1px solid black;
  border-color: black black red;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 40px 1% 0;
  padding: 0 2% 2%;
  width: 23%;
}
.
.
.
```

Listing 4.33 sets a border around the entire element and then changes the color of one of the sides. The second declaration doesn't overwrite the entire border declaration; instead, it has an effect only on the part that pertains to border color. So by starting with a more generic style and then adding another style that changes some specific element, you can often accomplish a lot of work in just a couple lines of CSS (Figure 4.33).

Figure 4.33: Now the bottom border is red.

Before moving on, let's remove the red border, and revert the padding change from Listing 4.32. The result should look like Listing 4.34.

Listing 4.34: *Reverting a couple of* `.bio-box` *styles.*

`index.html`

```
.
.
.
.bio-box {
  border: 1px solid black;
```

```
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 40px 1% 0;
  padding: 2%;
  width: 23%;
}
.
.
.
```

4.8.1 Border radius

The border can also have a *radius* set, which creates a box with rounded corners. To see how this works, add the CSS in Listing 4.35 to the styles targeting the social links on the test page.

Listing 4.35: *Adding a border-radius to an object to make rounded corners.*

index.html

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 0.5);
  border-radius: 10px;
  color: blue;
  display: inline-block;
  height: 36px;
  width: 36px;
}
.
.
.
```

The box should now have nicely rounded corners (Figure 4.34)!

Figure 4.34: Rounded corners on the social links.

4.8.2 Making circles

Want to see how to make circles using just HTML and CSS? The trick is to give the elements a set width and height, and then make the `border-radius` bigger than the width of the element, while also making sure the height and width of the element are equal (so that the "box" is a perfect square). Let's bump up the `border-radius` from Listing 4.33, and also give the `li` s in the `.social-list` a little margin (Listing 4.36).

Listing 4.36: *A very large value for* `border-radius` *makes a circle.*

index.html

```
.
.
.
/* SOCIAL STYLES */
.social-link {
  background: rgba(150, 150, 150, 0.5);
  border-radius: 99px;
  .
  .
  .
}
.social-list {
  list-style: none;
  padding: 0;
  text-align: center;
}
.social-list > li {
  display: inline-block;
  margin: 0 0.5em;
}
.
.
.
```

Look at those circles (Figure 4.35)!

Figure 4.35: Rounded boxes have become circles!

The links look a little weird with all that text, though, so let's change the text in the links in the hero and in the `.bio-box` es to the more compact values "Fb", "Tw", and "Gh" (for Facebook, Twitter, and GitHub, respectively). (We'll replace these with nice icons in Section 10). The result appears in Listing 4.37.

Listing 4.37: *Making the text a little shorter for the social links.*

index.html

```
.
.
.
<ul class="social-list">
  <li>
    <a href="http://example.com/" class="social-link">Fb</a>
  </li>
  <li>
    <a href="http://example.com/" class="social-link">Tw</a>
  </li>
  <li>
    <a href="http://example.com/" class="social-link">Gh</a>
  </li>
</ul>
.
.
.
<a href="http://twitter.com/mhartl" class="social-link">Tw</a>
.
.
.<a href="https://twitter.com/leedonahoe" class="social-link">Tw</a>
.
.
.
<a href="https://twitter.com/nickmerwin" class="social-link">Tw</a>
.
.
.
```

Save and refresh, and your links should look like Figure 4.36.

Figure 4.36: A little better.

Hmm... it still looks a little odd with the text up at the top, and down at the bottom of the page the text isn't even centered in the circle (in the `.full-hero` container it is inheriting a `text-align: center` style). Let's clean up the look and make sure that it stays the same regardless of usage.

We are going to add padding, and you may recall from Section 4.2 that we'll need to add in a `box-sizing: border-box` so that we make sure the padding doesn't change the dimensions of the element:

```
box-sizing: border-box;
padding-top: 0.85em;
```

Let's also change the color of the text, change the font, make it bold, align it to the center, and remove the underline with a new style called `text-decoration` (set to `none` to remove the default underline for links):

```
color: #fff;
font-family: helvetica, arial, sans;
font-weight: bold;
text-align: center;
text-decoration: none;
```

Finally, we'll set the font size using an `em` value (so that its size makes sense in the local context), add equal height and width, and set the *line height*, which we'll talk about more in a moment (Section 4.8.3):

```
font-size: 1em;
height: 2.5em;
line-height: 1;
width: 2.5em;
```

The equal height and width make the element a square, so that it will be a circle when `border-radius` is applied.

All together, those changes (plus a few more) are shown in (Listing 4.38). (To figure out the effect of the additional rules for yourself, apply the comment-out trick mentioned in Box 1.2.)

Listing 4.38: *Changes to almost all the properties of the social links.*

index.html

```
.
.
.
.social-link {
  background: rgba(150, 150, 150, 0.5);
  border-radius: 99px;
  box-sizing: border-box;
  color: #fff;
  display: inline-block;
  font-family: helvetica, arial, sans;
  font-size: 1rem;
  font-weight: bold;
  height: 2.5em;
  line-height: 1;
  padding-top: 0.85em;
  text-align: center;
  text-decoration: none;
  vertical-align: middle;
  width: 2.5em;
}
.
.
.
```

Those fully rounded and styled links look pretty great (Figure 4.37)!

Figure 4.37: Muuuuch better.

Of all the new styles that we used, the most potentially confusing ones are probably `font-family` and `vertical-align`.

Looking back at Listing 4.38, you might think that `vertical-align` is something that can position elements in the middle of other elements, but in reality it has an effect only on objects that are inline, or inline block, and it centers them only in relation to the line of text that they appear on. We'll be covering vertical alignment a couple of times later in the tutorial—using positioning in Section 5.8, and using a more modern method called *flexbox* in Chapter 7.

The font-family change in Listing 4.38 involves defining what is called a *font stack*, which is just a list of font options that the browser should try to use:

```
font-family: helvetica, arial, sans;
```

Sometimes fonts can't be loaded from the Internet, or aren't available on a user's computer, so you start with the specific font you want as the first font, and then add the names of alternate fonts (separated by commas). Different computers have different fonts installed by default, and users can also add their own.

For instance, Apple computers have a classic font called Helvetica (by "classic" we mean that it was designed in 1957 and there's even a documentary about it). Windows has a knockoff version of Helvetica called Arial (and designers hate it). To see what the two look like in comparison, check out Figure 4.38.[7]

Figure 4.38: Arial is considered a cheap knockoff of Helvetica and you probably shouldn't use it.

To find out which common fonts are available on which operating systems, you should consult a resource like CSS Font Stack. It is also possible to load your own custom fonts onto a user's computer, which is a great way to add to the unique visual branding of your site. We'll be covering such custom font-loading in Section cha:details-embed-links.

4.8.3 Line height

As seen briefly in Listing 4.38, another aspect of text design is the *line height*, which defines the space above and below text and other inline elements. Any text on your site that is multi-line should have the line-height increased to make reading easier. An ideal amount would be around 140% to 170%, depending on the font.

The `line-height` property works like `em` in that `1` equals `100%`, but there's no associated unit as with `em`, `px`, etc. For example, to arrange for the `.bio-copy` to have a line height equal to 150% of the base font size, we can set the `line-height` to `1.5`, as shown in Listing 4.39.

Listing 4.39: *Changing the line height for the* `.bio-copy`.

index.html

```
.
.
.
.bio-copy {
  font-size: 1em;
  line-height: 1.5;
}
.
```

The spacing between lines is now increased, which in some contexts can make the copy easier to read (Figure 4.39).

Figure 4.39: The result of increasing the line height.

4.8.4 Syncing up

Before moving on to Chapter 5, you can sync up your index page using the code in Listing 4.40.[8]

Listing 4.40: *The current index page.*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>
      /* GLOBAL STYLES */
      html, body {
        margin: 0;
        padding: 0;
      }
      h1 {
        font-size: 7vw;
        margin-top: 0;
      }
      a {
        color: #f00;
      }

      /* HERO STYLES */
      .full-hero {
        background-color: #c7dbfc;
        height: 50vh;
      }
```

```
    /* SOCIAL STYLES */
    .social-link {
      background: rgba(150, 150, 150, 0.5);
      border-radius: 99px;
      box-sizing: border-box;
      color: #fff;
      display: inline-block;
      font-family: helvetica, arial, sans;
      font-size: 1rem;
      font-weight: bold;
      height: 2.5em;
      line-height: 1;
      padding-top: 0.85em;
      text-align: center;
      text-decoration: none;
      vertical-align: middle;
      width: 2.5em;
    }
    .social-list {
      list-style: none;
      padding: 0;
      text-align: center;
    }
    .social-list > li {
      display: inline-block;
      margin: 0 0.5em;
    }

    /* BIO STYLES */
    .bio-wrapper {
      font-size: 24px;
      margin: auto;
      max-width: 960px;
      overflow: hidden;
    }
    .bio-box {
      border: 1px solid black;
      border-color: black black red;
      box-sizing: border-box;
      float: left;
      font-size: 1rem;
      margin: 40px 1% 0;
      padding: 0 2% 2%;
      width: 23%;
    }
    .bio-box h3 {
      color: #fff;
      font-size: 1.5em;
      margin: -40px 0 1em;
      text-align: center;
    }
    .bio-box img {
      width: 100%;
    }
    .bio-copy {
      font-size: 1em;
      line-height: 1.5;
    }
    .bio-copy a {
      color: green;
    }
  </style>
</head>
<body>
  <div class="full-hero hero-home">
    <h1>I'm an h1</h1>
    <ul class="social-list">
      <li>
        <a href="http://example.com/" class="social-link">Fb</a>
      </li>
```

```
      <li>
        <a href="http://example.com/" class="social-link">Tw</a>
      </li>
      <li>
        <a href="http://example.com/" class="social-link">Gh</a>
      </li>
    </ul>
  </div>
  <h2>I'm an h2</h2>
  <div class="bio-wrapper">
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Michael Hartl</h3>
      <a href="http://twitter.com/mhartl" class="social-link">Tw</a>
      <div class="bio-copy">
        <p>
          Known for his dazzling charm, rapier wit, and unrivaled humility,
          Michael is the creator of the
          <a href="http://railstutorial.org/">Ruby on Rails
          Tutorial</a> and principal author of the
          <a href="https://learnenough.com/">
          Learn Enough to Be Dangerous</a> introductory sequence.
        </p>

        <p>
          Michael is also notorious as the founder of
          <a href="http://tauday.com/">Tau Day</a> and author of
          <a href="http://tauday.com/tau-manifesto"><em>The Tau
          Manifesto</em></a>, but rumors that he's secretly a supervillain
          are slightly exaggerated.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Lee Donahoe</h3>
      <a href="https://twitter.com/leedonahoe" class="social-link">Tw</a>
      <div class="bio-copy">
        <p>
          When he's not literally swimming with sharks or hunting powder
          stashes on his snowboard, you can find Lee in front of his computer
          designing interfaces, doing front-end development, or writing some of
          the interface-related Learn Enough tutorials.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Nick Merwin</h3>
      <a href="https://twitter.com/nickmerwin" class="social-link">Tw</a>
      <div class="bio-copy">
        <p>
          You may have seen him shredding guitar live with Capital Cities on
          Jimmy Kimmel, Conan, or The Ellen Show, but rest assured Nick is a
          true nerd at heart. He's just as happy shredding well-spec'd lines
          of code from a tour bus as he is from his kitchen table.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <h3>??</h3>
      <p>
        The Future
      </p>
    </div>
  </div>
  <p>
    Learn Enough to Be Dangerous is a leader in the movement to teach the
    world <em>technical sophistication</em>, which includes both "hard
    skills" like coding, command lines, and version control, and "soft
     skills" like guessing keyboard shortcuts, Googling error messages, and
```

```
      knowing when to just reboot the darn thing.
    </p>
    <p>
      We believe there are <strong>at least a billion people</strong> who can
      benefit from learning technical sophistication, probably more. To join
      our movement,
      <a href="https://learnenough.com/#email_list">sign up for our official
      email list</a> now.
    </p>
    <h3>Background</h3>
    <p>
      Learn Enough to Be Dangerous is an outgrowth of the
      <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a> and the
      <a href="http://www.softcover.io/">Softcover publishing platform</a>.
      This page is part of the sample site for
      <a href="https://learnenough.com/css-tutorial"><em>Learn Enough CSS and
      Layout to Be Dangerous</em></a>, which teaches the basics of
      <strong>C</strong>ascading <strong>S</strong>tyle
      <strong>S</strong>heets, the language that allows web pages to be styled.
      Other related tutorials can be found at
      <a href="https://learnenough.com/">learnenough.com</a>.
    </p>
  </body>
</html>
```

4.8.5 Exercises

1. Change the border style of the `.bio-box` es from `solid` to `dashed`, then change it to `dotted`.
2. Use the `border-color` property, and short-hand values, to set the top and left of the `.bio-box` es to be invisible using either `transparent` or `rgba(0, 0, 0, 0)`, and the right and bottom sides to be black.

Chapter 5
Laying it all out

Now that we've got a good base of CSS knowledge, it's time to learn how to put everything together into a real website. This chapter and the next is where we really kick things into high gear, with material you're unlikely to see in any other CSS tutorial. To get started, our first step will be to transform our previous work into a more manageable set of *templates* and *page layouts* that can be easily reused and updated (in accordance with the DRY principle (Box 1.3)).

Along the way, we'll add more styling as a way to learn more complex aspects of CSS, while refining our design to be more suitable for use as a personal or business website. Combined with Chapter 6, the result will be a professional-grade example that shows a variety of aspects of modern site design.

5.1 Layout Basics

There are an infinite number of ways that you can design content layouts for the Web, but over the years certain conventions have become common to many sites, as shown in Figure 5.1. These

may include elements like a header that contains site navigation and a logo (which typically links to the homepage); a hero section (Section cha:values_view); paragraph-style content with optional asides; and a page footer containing repetition of some elements from the header as well as things like links to the About page, Contact, privacy policy, etc. These commonalities are the result of years of trial and error, and by incorporating such familiar elements into our site we help new visitors orient themselves and find what they're looking for.

Figure 5.1: Elements of a typical web page.

One thing you may notice from Figure 5.1 is that many elements, such as the header and footer, are the same (or nearly the same) on every page of our site. If we made each page by hand, that would mean we'd be repeating ourselves like crazy, and if we wanted to make a change updating all those pages would be a nightmare.

This is an issue we faced repeatedly in *Learn Enough HTML to Be Dangerous*, where we simply copied and pasted common elements like navigation links onto every separate page. Such repetition is a violation of the DRY principle (Box 1.3), and in *Learn Enough HTML to Be Dangerous* we promised to teach you how to use a *templating system* to solve this problem. In

this section, we'll fulfill this promise by installing and using the *Jekyll* static site generator to eliminate duplication in our layout.

5.2 Jekyll

When building a professional-grade website, it's essential to use a system capable of supporting templates to eliminate duplication. To accomplish this, we'll be using *Jekyll* (Figure 5.2), a free and open-source program for generating static websites (that is, sites that don't change from visit to visit).[1]

Figure 5.2: Not Jekyll and Hyde... rather, Jekyll the static site generator!

By learning Jekyll, you'll develop the skills needed to develop and deploy a real website—skills that are transferable to other static site builders (such as Middleman and Hugo) and to full-blown web frameworks (like Ruby on Rails). Learning the template language used by Jekyll (called *Liquid*) is also a valuable skill in itself, as Liquid is widely used in systems like the Shopify ecommerce platform.[2]

In addition to supporting templates, Jekyll also includes a bunch of other useful features:

- Write content in Markdown (the lightweight markup format we first discussed in *Learn Enough Text Editor to Be Dangerous*) in your text editor of choice.
- Write and preview your content on your site locally in your dev environment.
- Publish changes via Git (which also gives you an automatic off-site backup).
- Host your site for free on GitHub Pages.
- No database management.

Originally developed by GitHub cofounder Tom Preston-Werner, Jekyll is used by thousands of people around the world and is an industrial-strength tool for creating static websites. For example, the fundraising platform for U.S. President Barack Obama's 2012 reelection campaign, which handled 81,548,259 pageviews and raised over $250 million, was built using Jekyll:[3]

> *By using Jekyll, we managed to avoid the complexity that comes with most CMSes (databases, server configuration) and instead were able to focus on things like optimizing the UI and providing a better user experience. To work in this environment, the most a front-end engineer had to learn was the Liquid template language that Jekyll uses, and boy is that simple.*

5.2.1 Installing and running Jekyll

Jekyll is written in the Ruby programming language, and is distributed as a Ruby *gem*, or self-contained package of Ruby code. As a result, installing Jekyll is easy once you have a properly configured Ruby development environment.

If your system is not already configured as a dev environment, you should consult *Learn Enough Dev Environment to Be Dangerous* at this time. This step might prove challenging, especially if you decide to configure your native system, but in the long run the effort is well worth the reward.

Once you've got a working dev environment, you can install Jekyll using the `gem` command supplied by the RubyGems package manager, which comes for free with Ruby:

```
$ gem install jekyll -v 3.5.1
```

Although Jekyll is designed to work with a system of templates (Section 5.3), in fact it can work with a single file, such as our current `index.html`. To see how it works, we can run the Jekyll server in our project directory:

```
$ jekyll serve
```

If you're working on a native system or a virtual machine (as opposed to a cloud IDE), at this point the Jekyll app should be available at the URL http://localhost:4000, where localhost is the address of the local computer and 4000 is the *port number* (Box 5.1). The result should look something like Figure 5.3.

Figure 5.3: No more URL pointing to a file—you're running on a server now!

Box 5.1. Server ports

If you look at the URL for the Jekyll site, you'll notice that the URL ends in ":4000". That is the *server port*. If you end a URL with a colon followed by a number you are telling the browser to connect to that port on the server… so what does that mean?

You can think of server ports as being like individual phone numbers for different services that run on a computer. The default port number for the World Wide Web is port 80, so http://learnenough.com:80 is the same thing as http://learnenough.com, while the default port for a secure connection is 443, so https://learnenough.com:443 is the same thing as https://learnenough.com (with `https` in place of `http`). Other common port numbers include 21 (ftp), 22 (ssh), and 23 (telnet).

In the context of developing applications on a development machine, using port numbers allows us to solve the important problem of being able to run two or more apps simultaneously. Suppose, for example, that we wanted to run two different Jekyll websites on our development server. By default, both of them would be located at localhost:4000, but this would cause a conflict because the browser would have no way of knowing which site to serve when visiting that address. The solution is to add an extra piece of information, the port number, which allows the computer to distinguish between, say, app #1 running on localhost:4000 and app #2 running on localhost:4001.

As noted above, Jekyll's default server port is 4000, but we can set a different port number using the `--port` command-line option as follows:

```
$ jekyll serve --port 4001
```

To connect to this second server, we would then type localhost:4001 into our browser's address bar.

If you're using the cloud IDE suggested in *Learn Enough Dev Environment to Be Dangerous*, you'll have to pass options for the port number (Box 5.1) and host IP number when running the `jekyll` command:

```
$ jekyll serve --port $PORT --host $IP
```

Here `$PORT` and `$IP` should be typed in literally; they are *environment variables* provided by the cloud IDE to make the development site accessible on an external URL. Once the server is running, you can visit it by selecting Share and then clicking on the server URL, as shown in Figure 5.4. The result, apart from the browser URL, should be the same as for the local system shown in Figure 5.3. (For simplicity, in what follows we sometimes refer to localhost:4000, but users of the cloud IDE should use their personal URL instead. *Mutatis mutandis.*)

Figure 5.4: Sharing the URL on the cloud IDE.

After starting the Jekyll server, you should find a new folder in your project called `_site` (with a leading underscore):

```
$ ls
_site       index.html
```

This folder contains the output from the Jekyll server as it builds your site from the source files (currently just `index.html`).

The `_site` directory and all its contents are generated by Jekyll every time that a file is saved, and if you were to make any changes in the `_site` folder they will be automatically overwritten. As a result, you should never make changes in any of the `_site` files themselves—they would only be overwritten by Jekyll. There's nothing more frustrating than accidentally working on updates in an automatically generated folder, only to have your changes overwritten by an uncaring static site generator (Figure 5.5).[4]

Figure 5.5: TFW changes accidentally made in generated files get overwritten.

5.2.2 Exercises

1. Try starting Jekyll up on a non-standard port like `1234`.

5.3 Layouts, includes, and pages (oh my!)

One of the most powerful features of Jekyll is its ability to factor different parts of a website into reusable pieces. To accomplish this, Jekyll uses a system of folders and conventional names for files, along with a mini-language called *Liquid*. Originally developed by Tobi Lütke, cofounder of online store powerhouse Shopify,[5] Liquid is a system for adding content to a site using what are in effect simple computer programs.

Files inside a Jekyll project can either be static, meaning that they do not get processed by the Jekyll engine, or they can be dynamic and get constructed with Jekyll magic. (The *site* is still static because it consists of static files on the server, even if those files are generated dynamically by Jekyll. In other words, the site is static because the files don't change once they've been generated by Jekyll, so the results are the same for every visitor of the site.)

There are four main types of magic objects/files that the Jekyll engine can use in an automated way to build your site:

- layouts/layout templates
- includes
- pages/page templates
- posts

We'll discuss each of these in abstract terms for reference, but their exact uses won't become clear until we see some concrete examples starting in Section 5.4.

5.3.1 Layouts/layout templates

Anything in the special `_layouts` directory (which we'll create in Section 5.4) can have Jekyll magic, meaning those files get read by the engine looking for Liquid tags and other Jekyll formatting.

One of the key parts of many Jekyll pages is *frontmatter*, which is *metadata* at the top of an HTML file (in YAML format) that identifies the kind of layout to be used, a page-specific title, etc. A fairly complicated example might look like this, where the frontmatter is everything between the two triple-dashes `---`:

```
---
layout: post
title: This is the title of the post
postHero: images/shark.jpg
author: Me, Myself, and I
authorTwitter: http://twitter.com/mhartl
gravatar: https://gravatar.com/avatar/ffda7d145b83c4b118f982401f962ca6?s=150
postFooter: Additional information, and maybe a <a href="#">link or two</a>
---

<div>
  <p>Lorem ipsum dolor sit paragraph.</p>
<div>
```

In a simpler but still common example, the frontmatter identifies only the page layout template to be used when rendering the page:

```
---
layout: default
---

<div>
  <p>Lorem ipsum dolor sit paragraph.</p>
<div>
```

We'll see the effects of this sort of code starting in Section 5.4.

If there is no frontmatter in a layout file, then it is a true layout, and it needs to have a full HTML page structure. If there *is* frontmatter, then the file is a layout template that can be built into other layouts, and it doesn't need to have a full HTML page structure.

Layouts are often the most base-level objects, defining a standard page with a `DOCTYPE`, `html` / `head` / `body` tags, `meta` tags, stylesheet links, JavaScript, etc., and they usually pull in includes like a site header or site footer. You often need only one default layout for a site, but you can also use layout templates for things like blogs (Section 8.3).

Layouts have the special ability to load content, like posts, using a generic Liquid tag that looks like this: `{{ content }}`. We'll see a short example of this in an exercise (Section 5.6.3), and we'll apply it to our full site in Chapter 6.

5.3.2 Includes

Files in the `_includes` folder can have Jekyll magic even though they don't need frontmatter, and these files are always intended to be built into something else. Includes tend to be little snippets of a site that get repeated on many pages, such as the header and footer (Figure 5.1) or a standard set of social-media links. Includes will be covered in Section 5.6.

5.3.3 Pages/page templates

Any other HTML file in the project directory is a *page*. If there is no frontmatter in the file it is a *static page*, and Jekyll magic will not work (Liquid tags go unprocessed). If a page has

frontmatter, though, it will need to specify a layout, and then all the Jekyll magic will be available. We'll cover pages more in Section 6.

5.3.4 Posts, and post-type files

Posts are self-contained pieces of content, such as blog posts or product details, that are saved as files in the `_posts` directory. Content like blog posts are typically organized by date, while other post content (like product descriptions) are organized based on other attributes into *collections*. We'll discuss posts further in Chapter 8; collections are beyond the scope of this tutorial, but you can read about them in the Jekyll documentation on collections.

5.4 The layout file

Let's start playing around with a Jekyll layout by adapting our site into the framework. The end result of this section will be a page that looks exactly like the current `index.html` , but which is created in a way that will give us greater power and flexibility down the road. This includes getting a first taste of templates and frontmatter (which we'll cover in greater depth in Chapter 6).

This isn't how you would normally go about creating a site if you were starting from scratch. Layout files are usually pretty bare-bones (as we'll see in Section 6.1), and a more common development process is to create a spartan layout using the command `jekyll new` and then start doing the real work in the pages and includes. In our case, though, we've already done a lot of work in our single `index.html` file; using it as our initial layout means that, as we learn about different aspects of Jekyll, we can pull the parts we need out of the layout, thereby showing how a whole site can be sliced up and reassembled.

As we explained in Section 5.3, the Jekyll convention for layouts is to place these files in a directory called `_layouts` (with a leading underscore), which you should create in the root directory of your application (`repos/<username>.github.io`):

```
$ mkdir _layouts
```

Any HTML file in the `_layouts` directory can serve as a layout, so to get started we'll copy the existing `index.html` into the layouts directory to create a default layout:

```
$ cp index.html _layouts/default.html
```

At this point, your project files should look something like Figure 5.6.

Figure 5.6: Your files and directories should look like this.

To get our site back up and visible, replace the entire contents of `index.html` with the code shown in Listing 5.1.

Listing 5.1: *The site index with Jekyll frontmatter.*

index.html

```
---
layout: default
---
```

As mentioned in Section 5.3, the content in Listing 5.1 is known as the Jekyll *frontmatter*, and by adding it to the `index.html` file we've turned a static page into a Jekyll *page template*.

The frontmatter is the secret sauce that lets Jekyll know that it needs to read through an HTML page to see if it should process any of the content, and by specifying `layout: default` we've arranged for Jekyll to use `default.html` as the page layout. Because `default.html` is currently a fully self-contained page, the result of visiting http://localhost:4000 is to render our entire test page (Figure 5.3). In other words, Jekyll just takes the contents of `default.html` and inserts it into `index.html` .

As mentioned in Section 1.4, this sort of transformation, where we change the underlying code without changing the result, is known as *refactoring*. It may seem like we've done nothing, but we'll see in Section 5.6 how this new structure lets us slice and dice our website into reusable pieces.

5.4.1 Exercises

1. To see the way frontmatter affects how pages are built, delete the frontmatter in `index.html`, and write "Hello world." Save the file and refresh the page.

2. Revert your changes from Exercise 1, and change the layout to one called `test`. Then create a new file in the `_layout` directory called `test.html`, and add in some text like "Hello again, world."

3. In the root directory of your project, create a new file called `tested.html` and add some text in it like "For the third time, hello world!" Now in your browser go to http://localhost:4000/tested.html to see what happens.

5.5 CSS file and reset

Now that we've refactored our test page into a layout (`default.html`) and a page template (`index.html`), we're going to start the process of breaking our monolithic HTML/CSS file into its component parts. The first step is to create a standalone CSS file with a *reset* that eliminates troublesome browser defaults for margins, padding, etc. (Listing 3.18). Then we'll pull all the CSS out of the test site's `style` block and put it into the same external file.

To get started, create a new folder in the project directory called `css`, and then create a new file in that directory called `main.css`, either using the terminal like in Listing 5.2, or by just adding the folders and files in your text editor.

Listing 5.2: *Creating a new CSS folder and blank document in the terminal.*

```
$ mkdir css
$ touch css/main.css
```

You have to name your directory exactly `css`, because Jekyll automatically looks for CSS files in that location, but you can use whatever filename makes you happy for the actual CSS file.

After you've created the folder and file as in Listing 5.2, your project directory should look something like Figure 5.7.

Figure 5.7: The new `css` folder and `main.css` file.

Recall from the discussions in Section cha:values_em and Section cha:values_view that browsers have built-in default styling for many common elements. Those browser defaults can differ from browser to browser, and if we were to allow them to remain it would mean that many elements on the page would start with styles we didn't pick. No self-respecting and properly perfectionist developer (Figure 5.8) wants to leave the appearance of important elements up to the browser makers, so we'll apply a full *CSS reset* to create a blank slate for our designs.

Figure 5.8: Perfectionist kitteh hates having to restyle twice because of different browser defaults.

Recall that we created a mini-version of a CSS reset in Listing 3.18, where we reset the margin and padding for `html` and `body` tags. Now it's time to upgrade our site to use an industrial-strength reset. The resulting CSS may look intimidating, but don't worry—we're putting it in Listing 5.3 precisely so that you can copy and paste it without having to understand the details.

Listing 5.3: *A standard CSS reset.*

css/main.css

```
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend, table, caption,
tbody, tfoot, thead, tr, th, td, article, aside,
canvas, details, embed, figure, figcaption, footer,
header, hgroup, menu, nav, output, ruby, section,
summary, time, mark, audio, video {
  margin: 0;
  padding: 0;
  border: 0;
  font: inherit;
  vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
  display: block;
}
body {
  line-height: 1;
}
blockquote, q {
  quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
  content: '';
  content: none;
}
table {
  border-collapse: collapse;
  border-spacing: 0;
}
strong, b {
  font-weight: bold;
}
em, i {
  font-style: italic;
}
a img {
  border: none;
}
/* END RESET*/
```

Note that the CSS in Listing 5.3 doesn't need to be wrapped with the `style` tags the way the styles in the HTML file did; as we'll see in Listing 5.7, the browser understands from the link that everything inside the file is CSS.

We see in Listing 5.3 that most of the standard HTML elements get some sort of styling applied to them. The big block of selectors at the top is pretty much every HTML element in the spec

forced to have margin and padding set to zero, a border of zero, and told to inherit font styles. This might seem a little extreme to target every element, but when we are making a custom website there is no reason to leave browser defaults for things like margin, padding, and border in place—otherwise we could end up having to *undo* styling all over our stylesheet. It's better to undo a lot of stuff right off the bat, and then only add positive styling later on.

Also, don't think that the above reset styling is something set in stone (Figure 5.9). If later in your development career you find yourself adding the same styling to every (say) `table` tag on every site you design, it's probably best just to add that to your reset. As usual, the DRY principle applies (Box 1.3).

Figure 5.9: Reset rules aren't set in stone... or any other kind of tablet.

With the reset added, we're now in a position to move the custom CSS style developed so far in the tutorial into `main.css`. This involves first opening `default.html` and cutting all the CSS inside the `style` tag, leaving the tag empty (Listing 5.4).

Listing 5.4: *The default layout with CSS cut out.*

_layouts/default.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't Panic</title>
    <meta charset="utf-8">
    <style>

    </style>
  </head>
  <body>
  .
  .
  .
```

```
  </body>
</html>
```

Next, paste the CSS into `main.css` (possibly using something like Shift-Command-V, which pastes at the proper indentation level), and then delete the mini-reset shown in Listing 5.5 since it is now redundant. The result is shown in Listing 5.6.

Listing 5.5: *Delete these lines after pasting.*

css/main.css

```
html, body {
  margin: 0;
  padding: 0;
}
```

Listing 5.6: *The entire CSS file up to this point.*

css/main.css

```
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend, table, caption,
tbody, tfoot, thead, tr, th, td, article, aside,
canvas, details, embed, figure, figcaption, footer,
header, hgroup, menu, nav, output, ruby, section,
summary, time, mark, audio, video {
  margin: 0;
  padding: 0;
  border: 0;
  font: inherit;
  vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
  display: block;
}
body {
  line-height: 1;
}
blockquote, q {
  quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
  content: '';
  content: none;
}
table {
  border-collapse: collapse;
  border-spacing: 0;
}
strong, b {
  font-weight: bold;
}
em, i {
  font-style: italic;
}
a img {
  border: none;
}
/* END RESET*/
```

```
/* GLOBAL STYLES */
h1 {
  font-size: 7vw;
  margin-top: 0;
}
a {
  color: #f00;
}

/* HERO STYLES */
.full-hero {
  background-color: #c7dbfc;
  height: 50vh;
}

/* SOCIAL STYLES */
.social-link {
  background: rgba(150, 150, 150, 0.5);
  border-radius: 99px;
  box-sizing: border-box;
  color: #fff;
  display: inline-block;
  font-family: helvetica, arial, sans;
  font-size: 1rem;
  font-weight: bold;
  height: 2.5em;
  line-height: 1;
  padding-top: 0.85em;
  text-align: center;
  text-decoration: none;
  vertical-align: middle;
  width: 2.5em;
}
.social-list {
  list-style: none;
  padding: 0;
  text-align: center;
}
.social-list > li {
  display: inline-block;
  margin: 0 0.5em;
}

/* BIO STYLES */
.bio-wrapper {
  font-size: 24px;
  margin: auto;
  max-width: 960px;
  overflow: hidden;
}
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 40px 1% 0;
  padding: 2%;
  width: 23%;
}
.bio-box h3 {
  color: #fff;
  font-size: 1.5em;
  margin: -40px 0 1em;
  text-align: center;
}
.bio-box img {
  width: 100%;
}
.bio-copy {
  font-size: 1em;
  line-height: 1.5;
```

```
}
.bio-copy a {
  color: green;
}
```

As you can verify by refreshing the browser, the page is now completely unstyled (Figure 5.10).

Figure 5.10: It's been a long time since our site was this naked and unstyled.

To restore the styling, all we need to do is tell the layout page about `main.css`. The way to do this is to replace the `style` tags in the `head` section with a link to our stylesheet, as shown in Listing 5.7.

Listing 5.7: *Using a `link` tag to load `main.css`.*

_layouts/default.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page: Don't panic</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="/css/main.css">
  </head>
  .
  .
  .
```

The `link` tag in Listing 5.7 tells the browser that it will be loading a stylesheet (`rel` is short for "relationship"), and then specifies a URL (in this case an absolute one that looks at the site's root directory by starting the URL with a forward slash)[6] that leads to the file.

It's important to understand that using the `link` tag to load an external stylesheet has nothing to do with Jekyll; this general technique works even on hand-built websites that don't use any site builder. The stylesheet doesn't actually need to be local, either—theoretically, it can be anywhere on the Internet—but for our purposes we want to use a local file so that it's easy to make changes.

Now when you refresh the browser the styles should be properly applied, and the page will pretty much look how it did before our refactoring, although there will be some places where things don't look right because of the CSS reset (Figure 5.11).

Figure 5.11: Same old page, with some minor oddities.

Before moving on, let's make a few minor changes to prove that we know how to update styles via the CSS file. Ever since we started with this page, the fonts have looked a little... old-school. Let's add in a general style to the page `body` that will cascade down to every element on the page and change all body text to a nice, clean, sans-serif font (Listing 5.8).

Listing 5.8: *A good spot for this would be in the "Global Styles" section of the CSS file.*

css/main.css

```
/* GLOBAL STYLES */
body {
  font-family: helvetica, arial, sans;
}
.
.
.
.bio-copy {
  font-size: 1em;
  line-height: 1.5;
}
.
```

```
.
.
```

When you save your work and refresh the browser, everything should still look the way it did before, but with all-new fonts across the page (Figure 5.12).

Figure 5.12: Same old page, all new fonts.

Finally, in order to avoid the overlap between the bio box and social links, we'll change the CSS for the latter to be `display: block` with a margin, as shown in Listing 5.9.

Listing 5.9: *Fixing up the social link spacing.*

index.html

```
.
.
.
.bio-box img {
  width: 100%;
}
.bio-box .social-link {
  display: block;
  margin: 2em 0 1em;
}
.bio-copy {
  font-size: 1em;
}
.
.
.
```

The result appears in Figure 5.13.

Figure 5.13: Better spacing for the social links.

5.5.1 Exercises

1. Create a second CSS file in the `css` folder, and add a second link to this new CSS file in the head of the document (making sure that this second link comes after the original CSS link). In your new CSS file, add a style that changes the `.full-hero` background color to a red color of your choice. This shows that the order that stylesheets load affects which styles take priority.

2. Rename the new CSS to `reset.css`, and move the stylesheet link above the link to `main.css`. Now cut and paste the entire reset section from `main.css` into the new CSS file (overriding the style added in Exercise 1). Save everything and make sure that your test page looks the same in your browser. You've made your reset portable!

5.6 Includes intro: head and header

Now that we've factored the CSS into a separate file (and added a CSS reset), it's time to start slicing up the default page into reusable pieces. As discussed in Section 5.3, Jekyll provides *includes* to help us with this important task. (*Note*: In this context, the word "include" is used as a *noun*, which is not standard English but is standard in the world of static site builders. This usage also changes the pronunciation; the verb form is "in-CLUDE", but the noun form is "IN-clude".)[7]

Includes are supposed to be the smallest/most reusable snippets of site code. They are usually loaded into either layouts or templates, but in fact can be used anywhere on a site—you can even have includes call other includes (Figure 5.14). Since these snippets of code are intended to get dropped into the site almost anywhere, you should always try to make sure that any includes you create have code that is portable and self-contained.

Figure 5.14: You can put includes in includes, so your includes have includes.

Jekyll includes are located in a dedicated folder called `_includes` (as with `_layouts`, the underscore is important). Go ahead and create that folder now, together with a new file called `head.html` (Listing 5.10).

Listing 5.10: *Creating the includes folder and adding in a new file.*

```
$ mkdir _includes
$ touch _includes/head.html
```

At this point, your project folder should look something like Figure 5.15.

Figure 5.15: The project directory with added includes.

As you might have guessed, we're going to use `head.html` to hold the `head` tag and its contents. The way to do this is first to cut that content out of `default.html`, and then paste it into `head.html` (possibly using Shift-Command-V to paste with the right indentation), as shown in Listing 5.11.

Listing 5.11: *Moving* `head` *to its own file.*

_includes/head.html

```
<head>
  <title>Test Page: Don't Panic</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="/css/main.css">
</head>
```

To include the contents of `head.html` back into the `default.html` layout, we'll use our first example of the Liquid language mentioned in Section 5.3, which looks like this:

```
{% include head.html %}
```

Here `include` is a Liquid command to include the file in question (in this case, `head.html`). The special syntax `{% … %}` tells Jekyll to replace the contents of that line with the result of evaluating the code inside. Because Jekyll automatically knows to look in the `_includes` directory, the result will be to insert the contents of `head.html`.

Replacing the original `head` section with the corresponding Liquid snippet gives the code shown in Listing 5.12.

Listing 5.12: *Including the site head using Liquid.*

_layouts/default.html

```
<!DOCTYPE html>
<html>
  {% include head.html %}
  <body>
  .
  .
  .
```

After making these changes, you should refresh your browser to confirm that the page still works.

5.6.1 Page header: up top!

At the top of a typical web page, you will usually find some sort of site-level *navigation* that takes users from page to page on the site, and also includes site branding. This section is often referred to as the *site header* (Figure 5.16) (not to be confused with the `head` tag, which is the HTML header). Implementing such a header site-wide is a perfect application of Jekyll includes.

Figure 5.16: Some site headers from popular websites.

To get started, let's add a new Liquid tag to `header.html` (which we'll create in a moment) at the top of the `default.html` file, as shown in Listing 5.13.

Listing 5.13: *Including the header HTML.*

_layouts/default.html

```
<!DOCTYPE html>
<html>
  {% include head.html %}
  <body>
    {% include header.html %}
    <div class="full-hero hero-home">
      <h1>I'm an h1</h1>
      <ul class="social-list">
      .
      .
      .
```

Next, create a new blank document in the `_includes` folder called `header.html`:[8]

```
$ touch _includes/header.html
```

The header itself will use two *semantic elements* (i.e., elements that have meaning): `header` to contain the header and `nav` for the navigation links, which (as with the social links in Section 4.5) are organized as an unordered list `ul`. We'll also use the classes `"header"` and `"header-nav"` to make it easier to apply styles across a range of browsers (Box 5.2). The resulting code appears in Listing 5.14.

Listing 5.14: *The basic structure of our site header.*

_includes/header.html

```
<header class="header">
  <nav>
    <ul class="header-nav">
      <li><a href="/">Home</a></li>
      <li><a href="#">Nav 1</a></li>
      <li><a href="#">Nav 2</a></li>
      <li><a href="#">Nav 3</a></li>
    </ul>
  </nav>
  <a href="/" class="header-logo">Logo</a>
</header>
```

Save and refresh your browser and now you'll see your new site header (Figure 5.17). (We'll explain the placement of the logo in Section 5.6.2.)

Figure 5.17: Our not-very-attractive header.

Box 5.2. Style Note: Style HTML5 elements with classes

To ensure maximum backwards compatibility, it's not a good idea to target the newer HTML5 semantic elements like `header` and `nav` directly. There are inevitably going to be some users who visit your site on an old browser that doesn't support them—though luckily there are fewer such cases with each passing year.

When an old browser encounters new HTML tags, it sees them as regular `div` s, and any styles are ignored. To avoid this situation, it's better to give such elements classes, and then target your styles at the classes.

For example, we want to avoid styling `header` directly:

```
header {
  background: #000;
}
```

Instead, we'll give the `header` tag a class `"header"` (like in Listing 5.14), and then target that class (note the leading dot):

```
.header {
  background: #000;
}
```

This way, our styles will work even in older browsers.

5.6.2 Navigation and children

Next, let's style that ugly header!

The end goal for our design is to create a traditional sort of header, with a logo on the left-hand side that will send users back to the homepage, and site navigation at the top right. As a final step, we'll change the position of the header so that it will sit on top of content below it.

The first thing that we are going to do is move the navigation to the right and put the `li` s into a horizontal row by changing their display property to `inline-block` . The result, which we suggest inserting immediately after the global styles, appears in Listing 5.15.

Listing 5.15: *Adding header styles.*

`css/main.css`

```
.
.
.
/* GLOBAL STYLES */
.
.
.
/* HEADER STYLES */
.header-nav {
  float: right;
}
.header-nav > li {
  display: inline-block;
}
.
.
.
```

Note in Listing 5.15 that we've used the more advanced child selector `>` to target the `li` s (as discussed before in Box 4.1). That is to make sure that if we wanted to put a second level of links into the menu, only the direct children would be `inline-block` (which we will in fact do in Section cha:details-dropdown).

After saving and refreshing, you'll see that the menu has moved (Figure 5.18).

Figure 5.18: Navigation moved to the right and all in a line.

You might have wondered why the logo is *below* the navigational list in the Listing 5.14 even though it comes first when viewing the header from left to right. The reason is that we knew all along that we were going to float the navigation to the right side of the screen, and if the logo appeared before the navigation in the HTML order then the menu would start at the *bottom* of the logo. This is because even floating elements respect the line-height and position of normal block or inline elements that come before it, which in this case would lead to unwanted space around the logo. You can check this yourself by switching the positions of the logo and nav links; you'll see that the menu starts lower as a result (Figure 5.19).

Figure 5.19: Switching the logo to come first adds unwanted space.

Now let's add in some padding on the list items and make those links a little more stylish. We are going to add some padding to move the navigation away from the edges of the page:

```
padding: 5.5vh 60px 0 0;
```

We are also going to give each `li` in the navigation a bit of left margin so that it isn't bumping right up against its neighbor:

```
margin-left: 1em;
```

For the links themselves, we'll change the color and the size, make the font bold so that it is easier to read, get rid of the default link underlines (as is done in about 99% of site headers), and also automatically transform the text to be uppercase:

```
color: #000;
font-size: 0.8rem;
font-weight: bold;
text-decoration: none;
text-transform: uppercase;
```

Here we've used `#000` instead of `black`; as noted in Section 3.1.1, it's important to learn how to use these two interchangeably.

After adding the appropriate selectors, the styling changes look like Listing 5.16.

Listing 5.16: *Styling the navigational links.*

`css/main.css`

```
.
.
.
/* HEADER STYLES */
.
.
.
.header-nav {
  float: right;
  padding: 5.5vh 60px 0 0;
}
.header-nav > li {
  display: inline-block;
  margin-left: 1em;
}
.header-nav a {
  color: #000;
  font-size: 0.8rem;
  font-weight: bold;
  text-decoration: none;
  text-transform: uppercase;
}
.
.
.
```

Your page navigation should now look like Figure 5.20.

Figure 5.20: Navigational links are now a bit more stylish.

So how did we come up with those exact styles? The values came from just adding a couple of styling rules, and then tweaking the numbers until things looked good. Design isn't always a systematic process—often you just need to make changes and then play around with the numbers until you get something you like. When designing websites, there tends to be an extended period

of experimentation, so don't worry if it takes you time to get things right when you work on your own!

5.6.3 Exercises

1. You can load dynamic text into includes. To try this, add the extra code `{{ include.content }}` somewhere in your `header.html` include, and then in the layout change the include tag to `{% include header.html content="This is my sample note." %}`.

5.7 Advanced selectors

In order to add an extra bit of polish to the site header, we are going to introduce a few more advanced CSS selectors, and then we'll continue to add in more styling for the rest of our page. These advanced selectors include *pseudo-classes*, *first-child/last-child*, and *siblings*.

5.7.1 Pseudo-classes

It's always nice to have links do something when a user rolls over them, especially since we removed the underlines from the links in Listing 5.16. Those underlines on links are called *design affordances*, and they are there to give users the suggestion that something will happen if they move the cursor to the link and click.

Some people may argue that all links on a site should have some affordance that clearly marks them as something clickable, either with underlines or by making them look like buttons (HOLY WAR!!!). At this point in time though, the design convention of putting plain text links that don't have underlines (or some other special style) in a header is something that most Internet users are now accustomed to. You just know that the things at the top of the page are clickable.

Without underlines or other immediately visible affordances, though, it is important to show users a response to rolling over the link with their cursor (including on mobile (Box 5.3)). You really want people to know that they are interacting with an element that does something after they make an action.

Box 5.3. Style Note: Mobile hover consideration

Mobile users don't see rollover states, so you always need to be sure that the things you are designing will make sense to both mobile and desktop users. One way to do this is to make sure that you also style things so that there is a change when the link is actively clicked.

You might think that this would be something that happens automatically no matter what, but if you make any styling changes that alter links from their browser default, you will actually need to use the `:active` pseudo-class to define how you want a link to appear when someone interacts with it.

If you do end up removing all hints that something is clickable for your site on desktop, you might want to consider using a mobile media query to add in some hints specifically for mobile users. We'll be discussing this further in the context of *media queries* in Chapter 9.

All HTML links have a set of what are called *pseudo-classes* that allow developers to style different interactions with the link:

- `:hover` : styles what happens when a user rolls over the link (applies to any element, not just links).
- `:active` : styles what happens when a user clicks the link.
- `:visited` : styles what the link should look like if a users has already visited the linked page.

The way to add a pseudo-class to a style declaration is by combining the element or class name with the pseudo-class, like this:

```
.header-nav a:hover {
  color: #ed6e2f;
}
```

This use of the `:hover` pseudo-class arranges to change the color of the link when the user's mouse hovers over it. (For now we've just picked a random orange color that will stand out nicely against the blue background.)

We'll add a second change as well, which is to make the logo partially transparent on hover using the `opacity` property. The combined result appears in Listing 5.17.

Listing 5.17: *Adding hover states to the navigational links.*

```
 css/main.css

/* HEADER STYLES */
.
.
.
.header-nav a:hover,
.header-nav a:active {
  color: #ed6e2f;
}
.header-logo:hover,
.header-logo:active {
  opacity: 0.5;
}
.
.
.
```

Note that we've added the same styling to the `:active` pseudo-class in order to give mobile users feedback as well (as discussed in Box 5.3).

Save your styles and refresh and now the nav links will turn orange on rollover, and the logo will turn 50% transparent (the opacity style works like a decimal percentage), as shown in Figure 5.21.

Figure 5.21: Muuuuch better.

There are a bunch of other very useful pseudo-classes that are regularly used in designing layouts. We'll talk about some of these throughout the rest of this section, and we'll see further examples in Section cha:details-mobile-dropdown.

5.7.2 Exercises

1. Now that you've seen how to style rollovers, try styling the `.social-links` to have rollover states where the background color changes.
2. As stated in the section, psuedo-classes like `:hover` don't just apply to links. Try adding a hover state that changes the background color of the `.full-hero` element.

5.7.3 first-child

In order to indicate that the Home link in the navigation menu is particularly important, let's arrange for it always to be a different color from the others. We could do this with a separate class, but since Home is always going to be the first link in the menu we can target it using what is called the `first-child` pseudo-class. This pseudo-class applies the corresponding styles only

to the first child of the parent element. (There's also a `last-child` pseudo-class, which we'll use in Section cha:details-dropdown, and many others that are beyond the scope of this tutorial.)

Let's make the Home link work the *opposite* of the styling for the other links, so that it's orange by default and black on rollover. To use the first-child pseudo-class, we need to make sure that whatever we're targeting is contained in a wrapper, and that there is nothing else in the wrapper. That just means that when you are using the child pseudo-classes you need the elements to be inside of some other HTML element.

If there is anything like text, or an HTML element of a different type, between the top of the parent and the element you are trying to target, the first and last child pseudo-classes won't work, but in this case we *are* going to target the first `li` in `.header-nav` (Listing 5.18). The `ul` with the class `.header-nav` is our wrapper, and the `li` s are all children that can be targeted.

Listing 5.18: *Changing the appearance of just the first link.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.
.
.
.nav-links a:hover,
.nav-links a:active {
  color: #ed6e2f;
}
.header-nav > li:first-child a {
  color: #ed6e2f;
}
.header-nav > li:first-child a:hover {
  color: #000;
}
.
.
.
```

Note how specific we are in Listing 5.18: we're using the child selector to target only `li` s that are direct children of the `.header-nav` class. You don't technically need this level of precision, but later on we will add in a dropdown menu in the header (Section cha:details-dropdown), and if we target styles too generally then we'll make styling the dropdown difficult.

Now when you save and refresh the first link should look different (Figure 5.22).

Figure 5.22: Making the first nav link orange.

5.7.4 Exercises

1. We mentioned that there are other child selector types. Try out `:last-child` by changing the color of the link that is in the last `li` in the page header.

5.7.5 Siblings

Let's look at another two advanced selectors, and then after seeing how they work, we'll use one to add another little style detail to our site navigation. CSS supports two sibling selectors, both of which are written like the child selector `>` when making a declaration:

- the *adjacent sibling* `+` : selects a single element only if it is right next to the primary element in the declaration. For example, `h2 + p` selects a `p` tag only if it is immediately preceded by an `h2` tag.
- the *general sibling* `~` : selects all elements of the type in the declaration if they follow the primary element. For example, `h2 ~ p` applies to *all* `p` tags preceded by an `h2` tag.

Let's hop out of working on the header for a second to create an example to use with the sibling selectors. In your `default.html` file, replace the `h2` tag with the HTML from Listing 5.19.

Listing 5.19: *Replacing the* `h2` *and adding some text.*

```
_layouts/default.html

.
.
.
<h2>THE FOUNDERS</h2>
<p>
  Learn Enough to Be Dangerous was founded in 2015 by Michael Hartl, Lee Donahoe,
  and Nick Merwin. We believe that the kind of technical sophistication taught by
  the Learn Enough tutorials can benefit at least a billion people, and probably
```

```
  more.
</p>
<p>Test paragraph</p>
.
.
.
```

We can target the paragraph that directly follows the `h2` with the style shown in Listing 5.20.

Listing 5.20: *Adding an adjacent sibling selector.*

`css/main.css`

```
/* GLOBAL STYLES */
.
.
.
h2 + p {
  font-size: 0.8em;
  font-style: italic;
  margin: 1em auto 0;
  max-width: 70%;
  text-align: center;
}
.
.
.
```

Notice that only the first paragraph is styled (Figure 5.23).

Figure 5.23: Only the `p` immediately after the `h2` is styled.

Now if we change to the general sibling selector `~` as in Listing 5.21, both paragraphs will get styled (Figure 5.24).

Listing 5.21: *The general selector targets all elements that come after a specified element.*

`css/main.css`

```
.
.
.
h2 ~ p {
  font-size: 0.8em;
  font-style: italic;
  margin: 1em auto 0;
  max-width: 70%;
  text-align: center;
}
.
.
.
```

Figure 5.24: All `p` *tags after the* `h2` *are now styled the same.*

You may also have noticed from Figure 5.24 that the `p` s in the `.bio-box` es below aren't styled. That is because the sibling selectors don't pass styles to elements that are wrapped inside any other elements. They only work on elements inside the same parent.

Looking back to the header, we can use a sibling selector in the site header navigation to target all the `li` s after a first `li` , and add in a little extra styling to help visually separate the links using the styles in Listing 5.22. You might have seen something like this online: a little vertical line between navigational links to help separate them from other links in a list. Let's use a *sibling selector* to add some divider lines.

Listing 5.22: *Using the general sibling selector to add styling to the header navigation.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.
.
.
.header-nav > li {
  display: inline-block;
  margin-left: 1em;
}
.header-nav > li ~ li {
  border-left: 1px solid rgba(0, 0, 0, 0.3);
  padding-left: 1em;
}
.
.
.
```

The rule `.header-nav > li ~ li` in Listing 5.22 says to apply the subsequent rules to all `li` elements next to an initial `li` inside an element with class `".header-nav"` —in other words, every `li` in the menu after the first one. This way, the divider lines appear before every menu item except the first (Figure 5.25).

Figure 5.25: Menu divider lines.

Now that the navigation is fairly spiffy, let's turn our attention to the logo, which will give us a chance to learn a little bit about CSS positioning.

5.7.6 Exercises

1. What if you didn't use the tilde in Listing 5.22 (bar would be on all), but rather the adjacent sibling selector?

5.8 Positioning

In this section, we are going to take a look at how positioning works in CSS, focusing on the site logo, and then we'll finish off the header design. The whole CSS positioning topic can be a little tricky, and honestly there are people who work with CSS all the time who regularly get confused trying to get positioning to work right. So if this section seems long with lots of examples, just bear with us and work through it all—you'll find that understanding CSS positioning is an essential skill.

When you style an element's position, there are two basic possibilities:

1. Have the browser draw the element in its natural position in the normal flow of content on the page.
2. Remove the target from the flow of content and display it in a different place using directional styles—left, right, top, and bottom—and an additional dimension, the so-called `z-index`.

When an element is moved around out of its natural position with directional styles, it doesn't affect other elements in the document—it either covers them up or is hidden behind them. It

becomes like a ship cast adrift, torn free from its mooring on the page.

While it might be self-explanatory to move something left or right, or change its top or bottom position, you might not be familiar with the idea of a `z-index`. The `z-index` property (which can be any nonnegative number, 0 by default) determines whether an element is displayed above or below other elements, as in farther "into" the screen or farther "out" towards the viewer. It's an element's 3D position.

You can think of this like looking down at a big stack of papers—the higher the `z-index` number, the higher up the stack towards you the element is. A `z-index` of `0` would be the bottom-most piece of paper. We'll see a concrete example of the `z-index` in Section 5.9.

In order to change those directional styles, we first need to alter an element's `position` property. The `position` style in CSS can be given five different values (though one of them isn't really used). We'll start with one of the most common ones, *static*.

- `position: static` (Figure 5.26)
  - This is the default positioning of elements in the flow of content.
  - An element that has no position style set, or has `position: static`, will ignore directional styles like left, right, top, and bottom.

Figure 5.26: How `position: static` affects elements.

- `position: absolute` (Figure 5.27)
  - Positions the element at a specific place by taking it out of the document flow, either within a parent wrapper that has a `position:` value other than static, or (if there is no parent) then a specific place in the browser window. It is still a part of the page content, which means when you scroll the page, it moves with the content.
  - Also lets you define a `z-index` property.

- Because the element is removed from the document flow, the width or height is determined either by shrinking to the content inside, or by setting dimensions in CSS. It behaves kind of like an element set to `inline-block` .
- Causes any float that is set on the object to be ignored, so if you have both styles on an element you might as well delete the float.

Figure 5.27: How `position: absolute` affects elements.

- `position: relative` (Figure 5.27)
  - This is like static in that it respects the element's starting position in the flow of content, but it also allows directional styles to be applied that nudge the element away from the boundary with other elements.
  - It allows absolutely positioned items to be contained within, as though the relatively positioned element were a separate canvas. In other words, if an absolutely positioned element is inside a relatively positioned element, a style of `top: 0` would cause the absolutely positioned element to be drawn at the top of the relative position element rather than at the top of the page.
  - Also allows you to change the `z-index` of the element.

Figure 5.28: How `position: relative` *affects elements.*

- `position: fixed` (Figure 5.29)
  - Positions the element at a specific place within the browser window totally separate from the page content. When you scroll the page, it won't move.
  - Lets you set `z-index` .
  - Has the same need to have dimensions set as `position: absolute` ; otherwise, it will be the size of the content inside.
  - Also causes floats to be ignored.

Figure 5.29: How `position: fixed` *affects elements.*

- `position: inherit`
  - This is not very common so we aren't going to discuss other than to say it makes the element inherit the position from its parent.

Let's play around with some examples. First, let's add in some styles for the header to better see the boundaries and to give it dimensions (Listing 5.23).

Listing 5.23: *Added styles for the* `.header` *class.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.header {
  background-color: #aaa;
  height: 300px;
  width: 100%;
}
.
.
.
```

Let's now absolutely position the `.header-logo` and set it to `50px` from the bottom (Listing 5.24).

Listing 5.24: *Add an initial* `position: absolute` *to the logo.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.
.
.
.header-nav > li:first-child a:hover {
  color: #fff;
}
.header-logo {
  bottom: 50px;
  position: absolute;
}
.
.
.
```

Now save and refresh... where did the logo go (Figure 5.30)?

Figure 5.30: The parent container has no position style set.

The logo link ended up way at the bottom because the parent element that wraps the `.header-logo` doesn't have any `position` style applied. Also, if you scroll the page up and down you'll notice that the `.header-logo` still moves with the page. Let's constrain the logo to stay within the header by adding a position property, as shown in Listing 5.25.

Listing 5.25: *Setting a position other than static on the wrapper.*

css/main.css

```
.
.
.
.header {
  background-color: #aaa;
  height: 300px;
  position: relative;
  width: 100%;
}
.
.
.
```

With the `position` rule in Listing 5.25, the `.header-logo` will now be `50px` from the bottom of the gray header box, and any positions that we give to `.header-logo` will be determined based on the boundaries of the `.header` container (Figure 5.31). The way that the position is based off of the boundaries of the parent is what we meant when we said that setting a parent wrapper to `position: relative` made it like a separate canvas—everything inside that is absolutely positioned takes its place based on the dimensions of the parent.

Figure 5.31: The absolutely positioned `.header-logo` .

Note here that when an element is absolutely positioned, the directional styles don't add or subtract distance—setting `bottom: 50px` doesn't move it *toward* the bottom, but rather it sets the position `50px` *from* the bottom. So `right: 50px` puts the element `50px` from the right edge.

Negative positions work as well, and as long as the overflow of the parent wrapper isn't set to `hidden` , the absolutely positioned element will get placed outside the boundaries of the parent (Listing 5.26).

Listing 5.26: *Trying out negative positioning on our object.*

css/main.css

```
.
.
.
.header-logo {
  bottom: -50px;
  position: absolute;
  right: 50px;
}
.
.
.
```

After adding that style and refreshing your browser, the logo should be in a position similar to what is shown in Figure 5.32.

Figure 5.32: Positioning the logo on the right-hand side.

You might be asking, "Well, what happens if I set both a top *and* bottom, or a left *and* right?" The answer is that, for whatever reasons, the top and left properties will take priority and the bottom and right will be ignored.

Another thing to consider is when you set a position property, you are manipulating elements and messing around with the natural page flow, which means that it is possible to cause misalignments. So if you add `left: 200px` to the `.header`, the width of the element (which is 100%) isn't recalculated. Instead, the entire `.header` box is pushed over by 200px, and your browser window will have horizontal scroll bars and look broken (Figure 5.33).

Figure 5.33: This sort of thing looks sloppy.

You have to be careful!

While we are still just playing around in the positioning sandbox, we should take a look at ways to deal with a situation that comes up any time positioning in CSS is discussed: how do you center an absolutely positioned object horizontally and vertically in a way that allows the object to be any size... and allows the wrapper to be any size?

Let's first look at an old method where the object that we are centering has a set height and width —centering this is easy. Give the logo a width and height, remove the old positioning, and change the background to better see the object (Listing 5.27).

Listing 5.27: *Adding height and width dimensions to the logo.*

css/main.css

```
.
.
.
.header-logo {
  background-color: #000;
  height: 110px;
  position: absolute;
  width: 110px;
}
.
.
.
```

Now let's center it.

You might think that centering the element would be as simple as giving the `.header-logo` class a style of `left: 50%` and `top: 50%` —that should put it in the middle, both horizontally and vertically, right (Listing 5.28)?

Listing 5.28: *Positioning the .header-logo in the center?*

css/main.css

```
.
.
.
.header-logo {
  background-color: #000;
  height: 110px;
  left: 50%;
  position: absolute;
  top: 50%;
  width: 110px;
}
.
.
.
```

Well, no, the reason this didn't work is that when the browser positions an object, it calculates the distance using the same-named edge—so when you apply `top: 50%` , it moves the top edge (not the center point) of `.header-logo` 50% away from the top of `.header` ; similarly, applying `left: 50%` tells the browser to move the left edge 50% away from the left of `.header` . The result is that the object we are trying to position is off-center by half of its width and height (Figure 5.34).

Figure 5.34: The red box is the expected position if centered vertically and horizontally.

How do we solve this and get our object in the actual center? The older method mentioned above was to use a negative margin (Section cha:box_margin-negative) to move the object up and left. This only works if you know the size of the object, though, since trying to use something like a percentage would move the object based on the size of the parent (recall from Section cha:values_percent that percentage values are based on the size of the parent object). Since the height and width of the box are `110px`, half of that is `55px` (Listing 5.29).

Listing 5.29: *Adding in the negative margins to position the black box in the right spot.*

css/main.css

```
.
.
.
.header-logo {
  background-color: #000;
  height: 110px;
  left: 50%;
  margin: -55px 0 0 -55px;
  position: absolute;
  top: 50%;
  width: 110px;
}
.
.
.
```

That works just fine, but you'd always be limiting yourself to centering only objects with fixed dimensions (Figure 5.35).

Figure 5.35: Negative margins worked!

If you wanted to make a slightly bigger (or smaller) centered object, you'd have to recalculate sizes and margins, and then make changes to your CSS. That's too much work, and it wouldn't

work at all with dynamically sized elements. Thankfully there is a better, relatively new CSS style called `transform` that can help. The `transform` property allows developers to do all sorts of crazy things like move objects around, rotate them, or simulate three-dimensional movement.

The upside for centering objects is that this new style calculates all these movements based on the object itself. So if we move it 50% to the left using `transform`, the browser looks at the object's width, and then moves it to the left 50% of its own width, not the width of the parent.

The actual style declaration looks like this: `transform: translate(x, y)` —where `x` is replaced by the distance along the x-axis (left is negative, right is positive), and the same for the y-axis (up is negative, down positive). So, to move our object left and up half its width and height, we'd add the transform style like you see in Listing 5.30 (make sure to remove the margin styling that we added in the Listing 5.29).

Listing 5.30: *Adding in the negative margins to position the black box in the right spot.*

css/main.css

```
.
.
.
.header-logo {
  background-color: #000;
  height: 110px;
  left: 50%;
  position: absolute;
  top: 50%;
  transform: translate(-50%, -50%);
  width: 110px;
}
.
.
.
```

Now when you save your work and refresh the browser you'll have a black box in the center of the gray header. It doesn't matter what dimensions you give for either the `.header-logo` or `.header` —you'll always have a vertically and horizontally centered object. To try it out, delete the height and width that we gave the `.header-logo` (Listing 5.31).

Listing 5.31: *Deleting the fixed width and height.*

css/main.css

```
.
.
.
.header-logo {
  background-color: #000;
  left: 50%;
  position: absolute;
  top: 50%;
  transform: translate(-50%, -50%);
}
.
.
.
```

When you save and refresh your browser, the now-smaller box will still be centered vertically and horizontally (Figure 5.36).

Figure 5.36: No matter what size the object, it stays right in the center.

5.8.1 A real logo

All right, enough positioning playtime. Let's get back to making this site look good by putting an actual logo in that `.header-logo`. In your project directory, add a new folder called `images` (Figure 5.37):

```
$ mkdir images
```

Figure 5.37: New images folder in your project directory.

Then use this `curl` command to grab the logo image off the Learn Enough servers:

```
$ curl -o images/logo.png -OL cdn.learnenough.com/le-css/logo.png
```

Now let's put the image into the `header.html` (Listing 5.32). The result appears in Figure 5.38.

Listing 5.32: *Replacing the word logo with a logo image.*

_includes/header.html

```
<header class="header">
  <nav>
    <ul class="header-nav">
      <li><a href="/">Home</a></li>
      <li><a href="#">Nav 1</a></li>
      <li><a href="#">Nav 2</a></li>
      <li><a href="#">Nav 3</a></li>
    </ul>
  </nav>
  <a href="/" class="header-logo">
    <img src="/images/logo.png" alt="Learn Enough">
  </a>
</header>
```

Figure 5.38: The initial (sub-optimal) logo placed on the page.

Now we are going to make a whole lot of changes to whip this part of the site into shape. As in Section 5.6.2, we aren't going to go through and give a reason why each value is the exact number we chose. Styling a section of a site is a non-linear process at times, and you'll likely need to experiment a lot.

First, we are going to make the header background color black and any text in the header white as follows:

```
.header {
  background-color: #000;
  color: #fff;
}
```

That's also going to require that we change the color of the links, as well as the rollover color for the first-child link in the navigation:

```
.header-nav > li:first-child a:hover {
  color: #fff;
}
```

We'll also need to change the background color of our little divider lines so that it is partially transparent white instead of partially transparent black:

```
border-left: 1px solid rgba(255, 255, 255, 0.3);
```

Then we are going to move the `.header-logo` into the top left, and shrink the image a bit:

```
.header-logo {
  background-color: #000;
  box-sizing: border-box;
  display: block;
  height: 10vh;
  padding-top: 10px;
  position: relative;
  text-align: center;
  width: 10vh;
}
.header-logo img {
  width: 4.3vh;
}
```

We chose `10vh` for the size of the link, and for the image we set the width to be 4.3% of the height of the container (`4.3vh`). We got those values after playing around with different numbers, and settling on this size for a balance of readability while not taking up too much space.

You'll notice that most of the sizing styles are on the link that wraps the image and not on the image itself. The reason we did that was so that if there is a problem downloading the image, or a delay, there is still a nice big clickable link in the header.

Putting everything together gives us Listing 5.33, which includes all the styling for the site header so far.

Listing 5.33: *Changing up the styling for the header and logo.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.header {
  background-color: #000;
  color: #fff;
}
.header-logo {
  background-color: #000;
  box-sizing: border-box;
  display: block;
  height: 10vh;
  padding-top: 10px;
  position: relative;
  text-align: center;
  width: 10vh;
}
.header-logo:hover,
.header-logo:active {
  background-color: #ed6e2f;
}
.header-logo img {
  width: 4.3vh;
}
.header-nav {
  float: right;
  padding: 5.5vh 60px 0 0;
}
.header-nav > li {
  display: inline-block;
  margin-left: 1em;
}
.header-nav > li ~ li {
  border-left: 1px solid rgba(255, 255, 255, 0.3);
```

```
  padding-left: 1em;
}
.header-nav a {
  color: #fff;
  font-size: 0.8rem;
  font-weight: bold;
  text-decoration: none;
  text-transform: uppercase;
}
.header-nav a:hover,
.header-nav a:active {
  color: #ed6e2f;
}
.header-nav > li:first-child a {
  color: #ed6e2f;
}
.header-nav > li:first-child a:hover {
  color: #fff;
}
.
.
.
```

Save and refresh, and your header should look like Figure 5.39. That logo's lookin' sharp!

Figure 5.39: The header, now styled.

5.8.2 Exercises

1. Try moving the `ul` that contains the social links to the bottom left corner of the `.full-hero` using the positioning rules you've learned. What changes are you going to need to make to `.full-hero` to allow the social links to remain inside?

2. To see why we gave dimensional styling and an `alt` tag to our image, try removing the image source link to simulate the browser not finding the file.

5.9 Fixed header

You may have noticed the recent design trend where the header sticks to the top of the screen as you scroll down the page. This is called a *fixed header*—the header is styled to use `position: fixed` to take the header entirely out of the page content and stick it to the top of the user's browser. If your site has a bunch of different sections that your users need to navigate to, a fixed header can be a good solution to keep them from getting annoyed that they always have to scroll to the top to do something new.

The way to implement a fixed header is to change the positioning of the header to `fixed` while specifying a `z-index` for the header. Recall from the beginning of Section 5.8 that the z-index determines whether an element is drawn in front or behind other elements. We'll want to give our header a large value for z-index, which will force the browser to draw the element above other elements (i.e., closer to the user using our stack-of-paper analogy).

The styles to change the positioning value and set a z-index are shown in Listing 5.34.

Listing 5.34: *Fixing the header's position means that content will now scroll under it.*

css/main.css

```
.
.
.
.header {
  background-color: #000;
  color: #fff;
  position: fixed;
  width: 100%;
  z-index: 20;
}
.
.
.
```

When you check the work in your browser, you'll find that the header is now pinned to the top of the screen, and when you scroll, all the content will scroll underneath (Figure 5.40).

Figure 5.40: A fixed header—you can see that it is covering content now.

The resulting black bar at the top looks cool, but what if we were to put a border around the entire page? It could look interesting to have a dark area around the whole site to frame the content. We can arrange for this with the styling shown in Listing 5.35.

Listing 5.35: *Just for fun, let's put a border around the entire site.*

css/main.css

```
.
.
.
/* GLOBAL STYLES */
html {
  box-shadow: 0 0 0 30px #000 inset;
  padding: 0 30px;
}
.
.
.
```

Listing 5.35 introduces the `box-shadow` style, which is a relatively new CSS style that lets you add drop shadows to HTML elements, and the declaration that we added is a shorthand for `box-shadow: x-axis y-axis blur size color inset`. We aren't going to go any deeper into it, but if you want to play around with box shadows there are a number of sites that let you fiddle with the settings, such as CSSmatic box shadow.

After applying the code in Listing 5.35, your page should now look like Figure 5.41.

Figure 5.41: Box shadow inset around the entire page. Nifty.

After saving and refreshing, you might have noticed that the logo in the header now looks a little off since it isn't right up in the corner anymore. This is because we increased the padding on the entire site by `30px` for the black border. Let's use a negative value (`-30px`) on the positioning to get it back in place, as shown in Listing 5.36.

Listing 5.36: *Using negative value to move the logo back into place.*

css/main.css

```
.
.
.
.header-logo {
  background-color: #000;
  box-sizing: border-box;
  display: block;
  height: 10vh;
  left: -30px;
  padding-top: 10px;
  position: relative;
  text-align: center;
  width: 10vh;
}
.
.
.
```

The fixed final header should now look like Figure 5.42 (shown as it should appear with the mouse cursor on the logo, making it orange).

Figure 5.42: A completed page header.

One thing you might have noticed is that after adding fixed positioning to the header, the big `h1` text in the hero is covered. We'll tackle this issue in Section 6.2.

Now that we've got the header squared away, let's turn our attention to the other end of the site.

5.9.1 Exercises

1. To see why it is important to define the z-index of the header, try setting the value to `1`, and then add styles to the `.social-list` class to set `position: relative` and `z-index: 40`. Then scroll the page.

5.10 A footer, and includes in includes

After creating and styling a site header, a natural next step is to style the page footer. This is the navigational/informational section that can be found at the bottom of a site (Figure 5.43).

Figure 5.43: A refresher on the elements of a typical web page, including a page footer.

Often, the footer is a partial replication of the navigational elements from the header (just styled in a slightly different way), but many sites add to that a bunch of other content—everything from store locations and hours to additional content links.

Since the footer is found at the end of the page, and contains ancillary information, you don't really need to worry about space (there's plenty of room at the bottom!). What we mean by that is that you can think of the footer as extra space, where users aren't *required* to see everything there. Many sites, such as Amazon, have a lot of content in a giant footer at the bottom of the page (Figure 5.44).

Figure 5.44: A giant footer.

We'll start by creating a new `footer.html` file inside the `_includes` folder:

```
$ touch _includes/footer.html
```

Next, we'll add some HTML. We're going to wrap the footer in another HTML5 semantic tag, the `footer` tag. As with the `header` tag, this is a semantic element that works just like a standard `div`, but gives automated site readers (such as web spiders and screen readers for the visually impaired) a better idea of what the purpose is of the content inside. We are also going to add in a logo link similar to the one in the header. The result appears in Listing 5.37.

Listing 5.37: *Adding in the basic footer structure.*

_includes/footer.html

```
<footer class="footer">
  <a class="footer-logo" href="/">
    <img src="/images/logo.png" alt="Learn Enough"/>
  </a>
  <h3>Learn Enough <span>to Be Dangerous</span></h3>
</footer>
```

To include the footer in the default layout, we'll follow the model from Listing 5.12 and use Liquid to insert the contents of `footer.html` just before the closing `body` tag in `default.html` (Listing 5.38).

Listing 5.38: *Add in the Liquid tag to the default layout*

_layouts/default.html

```
.
.
```

```
    .
    </p>
    {% include footer.html %}
  </body>
</html>
```

Now let's add some styling as well. We'll give the footer a black background, like the header, and also we'll give it some padding. We'll make sure that the content inside is easy to read by using `vh` units, which causes our padding to take up a large portion of the screen:

```
background-color: #000;
padding: 10vh 0 15vh;
```

We'll also constrain the size of the logo so that it isn't a giant image, and style the `h3` and the span that is inside of it (just to add a little design detail to give some of the text a different color). All together the footer styling looks like Listing 5.39.

Listing 5.39: *The initial styles for the footer.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.
.
.
/* FOOTER STYLES */
.footer {
  background-color: #000;
  padding: 10vh 0 15vh;
  text-align: center;
}
.footer-logo img {
  width: 50px;
}
.footer h3 {
  color: #fff;
  padding-top: 1.5em;
  text-transform: uppercase;
}
.footer h3 span {
  color: #aaa;
}

/* HERO STYLES */
.
.
.
```

Save and refresh, and the result should appear as in Figure 5.45.

Figure 5.45: The first stab at the footer is looking pretty good.

...and not too bad!

But let's make it a little more useful and also add in the navigational links from the header. You could just copy and paste the HTML from the header, but if you added a new page you'd have to edit your navigation in two spots... we hope the mere suggestion of that is making your programmer's itch flare up again. Since those nav links are always going to be the same in both the header and the footer, we can create a new include to include in includes (thereby fulfilling the promise from Figure 5.14—it wasn't (just) a joke!).

We don't want to take the outer `ul` from the Listing 5.14 since it has a `header-nav` class applied to it (well, you *could* add that in the include, then unstyle all the header styles, and then restyle to fit the footer—but that would be a lot of unnecessary work). So the content of our new include will just be the `li` s and the links—in other words, the content that definitely needs to be repeated.

To eliminate repetition in the links, let's create a new file in the `_includes` directory and name it `nav-links.html` :

```
$ touch _includes/nav-links.html
```

Then cut the `li` s and links out of the `.header-nav` and paste them into the new include, as shown in Listing 5.40.

Listing 5.40: *We've cut and pasted in the `li` s and links.*

_includes/nav-links.html

```
<li><a href="/">Home</a></li>
<li><a href="">Nav 1</a></li>
<li><a href="">Nav 2</a></li>
<li><a href="">Nav 3</a></li>
```

With the code in Listing 5.40, we can replace the links in the header file with a Liquid tag, as shown in Listing 5.41.

Listing 5.41: *Updating the header with an include and a second class.*

_includes/header.html

```
.
.
.
<ul class="header-nav nav-links">
  {% include nav-links.html %}
</ul>
.
.
.
```

Note that we've also added a `.nav-links` class in Listing 5.41 so we can add styling to the links that will be shared between the header and footer. Before, we were targeting and styling the links using the class `.header-nav` (introduced in Listing 5.14), but now that the links are going to be in multiple places that isn't a good name to use to target the styling common to both the header and the footer.

Now that we've factored the nav links into a separate include, let's add them to the navigation section in the footer. In order to allow footer-specific styling, we'll also add a `footer-nav` class (in analogy with the header's `header-nav` class), as well as the general `nav-links` class added in Listing 5.41. The result appears in Listing 5.42.

Listing 5.42: *The new Liquid tag to load the links in the footer.*

_includes/footer.html

```
<footer class="footer">
  <a class="footer-logo" href="/">
    <img src="/images/logo.png" alt="Learn Enough"/>
  </a>
  <nav>
    <ul class="footer-nav nav-links">
      {% include nav-links.html %}
    </ul>
  </nav>
  <h3>Learn Enough <span>to Be Dangerous</span></h3>
</footer>
```

Now let's add some styling. First, we should move some of the styles that before were defined on `.header-nav a` over to `.nav-links a`, and change the class that is targeting the `:hover` and `:active` states from `.header-nav` to `.nav-link`, as in Listing 5.43.

Listing 5.43: *Moving link styling into a new* `.nav-links` *class.*

css/main.css

```
.
.
.
.header-nav a {
  color: #fff;
}
.nav-links a {
  font-size: 0.8rem;
  font-weight: bold;
  text-decoration: none;
  text-transform: uppercase;
}
.nav-links a:hover,
.nav-links a:active {
  color: #ed6e2f;
}
.
.
.
```

Again, the idea is that we want navigational links to look similar between the header and footer, and then for any changes that are specific to one location or the other by targeting the links using either the `.header-nav` or the `.footer-nav` class.

Finally, we'll add footer-specific styles, as shown in Listing 5.44.

Listing 5.44: *New styling for footer navigation and links.*

`css/main.css`

```
.
.
.
/* FOOTER STYLES */
.
.
.
.footer-nav li {
  display: inline-block;
  margin: 2em 1em 0;
}
.footer-nav a {
  color: #ccc;
}
.
.
.
```

When you save and refresh, you'll have a nice header and footer, both pulling their navigational links from the same place (Figure 5.46).

Figure 5.46: Styled header and footer with nav links from an include.

If you want to double-check and sync up all your styles, Listing 5.45 has the current state of the CSS declarations for the site.

Listing 5.45: *The full header and footer styles.*

css/main.css

```
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend, table, caption,
tbody, tfoot, thead, tr, th, td, article, aside,
canvas, details, embed, figure, figcaption, footer,
header, hgroup, menu, nav, output, ruby, section,
summary, time, mark, audio, video {
  margin: 0;
  padding: 0;
  border: 0;
  font: inherit;
  vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
  display: block;
}
body {
  line-height: 1;
}
blockquote, q {
  quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
```

```
  content: '';
  content: none;
}
table {
  border-collapse: collapse;
  border-spacing: 0;
}
strong, b {
  font-weight: bold;
}
em, i {
  font-style: italic;
}
a img {
  border: none;
}
/* END RESET*/

/* GLOBAL STYLES */
html {
  box-shadow: 0 0 0 30px #000 inset;
  padding: 0 30px;
}
body {
  font-family: helvetica, arial, sans;
}
h1 {
  font-size: 7vw;
  margin-top: 0;
}
a {
  color: #f00;
}
h2 ~ p {
  font-size: 0.8em;
  font-style: italic;
  margin: 1em auto 0;
  max-width: 70%;
  text-align: center;
}

/* HEADER STYLES */
.header {
  background-color: #000;
  color: #fff;
  position: fixed;
  width: 100%;
  z-index: 20;
}
.header-logo {
  background-color: #000;
  box-sizing: border-box;
  display: block;
  height: 10vh;
  left: -30px;
  padding-top: 10px;
  position: relative;
  text-align: center;
  width: 10vh;
}
.header-logo:hover,
.header-logo:active {
  background-color: #ed6e2f;
}
.header-logo img {
  width: 4.3vh;
}
.header-nav {
  float: right;
  padding: 5.5vh 60px 0 0;
}
```

```
.header-nav > li {
  display: inline-block;
  margin-left: 1em;
}
.header-nav > li ~ li {
  border-left: 1px solid rgba(255, 255, 255, 0.3);
  padding-left: 1em;
}
.header-nav a {
  color: #fff;
}
.nav-links a {
  font-size: 0.8rem;
  font-weight: bold;
  text-decoration: none;
  text-transform: uppercase;
}
.nav-links a:hover,
.nav-links a:active  {
  color: #ed6e2f;
}
.header-nav > li:first-child a {
  color: #ed6e2f;
}
.header-nav > li:first-child a:hover {
  color: #fff;
}

/* FOOTER STYLES */
.footer {
  background-color: #000;
  padding: 10vh 0 15vh;
  text-align: center;
}
.footer-logo img {
  width: 50px;
}
.footer h3 {
  color: #fff;
  padding-top: 1.5em;
  text-transform: uppercase;
}
.footer h3 span {
  color: #aaa;
}
.footer-nav li {
  display: inline-block;
  margin: 2em 1em 0;
}
.footer-nav a {
  color: #ccc;
}

/* HERO STYLES */
.full-hero {
  background-color: #c7dbfc;
  height: 50vh;
}

/* SOCIAL STYLES */
.social-list {
  list-style: none;
  padding: 0;
  text-align: center;
}
.social-link {
  background: rgba(150, 150, 150, 0.5);
  border-radius: 99px;
  box-sizing: border-box;
  color: #fff;
  display: inline-block;
```

```
  font-family: helvetica, arial, sans;
  font-size: 1rem;
  font-weight: bold;
  height: 2.5em;
  line-height: 1;
  padding-top: 0.85em;
  text-align: center;
  text-decoration: none;
  vertical-align: middle;
  width: 2.5em;
}
.social-list > li {
  display: inline-block;
  margin: 0 0.5em;
}

/* BIO STYLES */
.bio-wrapper {
  font-size: 24px;
  margin: auto;
  max-width: 960px;
  overflow: hidden;
}
.bio-box {
  border: 1px solid black;
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 40px 1% 0;
  padding: 2%;
  width: 23%;
}
.bio-box h3 {
  color: #fff;
  font-size: 1.5em;
  margin: -40px 0 1em;
  text-align: center;
}
.bio-box img {
  width: 100%;
}
.bio-box .social-link {
  display: block;
  margin: 2em 0 1em;
}
.bio-copy {
  font-size: 1em;
}
.bio-copy a {
  color: green;
}
```

Finally, in case you haven't been doing your own Git commits and deploys, now would be a good time to do one:

```
$ git add -A
$ git commit -m "Finish initial layout"
```

You'll discover that GitHub Pages is fully Jekyll-aware, and automatically generates and displays the site based on the contents of the repository—free static site hosting!

5.10.1 Exercises

1. **(challenging)** In the same manner that we just made the header links modular, first create a new include that makes the social links in the hero into an include that can be put on other places on the site. Then use the correct include tag to put it back where it originally was, and also a second include that builds the social links into a new `ul` in the footer.

Chapter 6
Page templates and frontmatter

You might have noticed that we finished Chapter 5 with a big loose end: the default "layout" `default.html` contains all of the content from our main index page. This makes it effectively useless for reuse, since as currently constructed the layout would include the same home page content on every page of our site. In this chapter, we'll tie up this loose end by learning how to insert content dynamically, thereby making our layout a truly reusable template.

What we want is to be able to keep the site-wide elements like the header and footer while varying the content in between (the part shown in blue in Figure 6.1). Our method will be to use Jekyll page templates to further simplify our site's structure and refine its styling, with a focus on developing unique content for each page. Along the way, we'll learn a new Liquid command for inserting content (Section 6.1), style the hero section (Section 6.2), learn some more advanced selectors (Section 6.3), and add pages other than the default index (Section 6.4).

Figure 6.1: The content is sandwiched between the header and footer.

6.1 Template content

As mentioned in Section 5.4, the current state of `default.html` doesn't represent the standard way of using a Jekyll layout. The reason is that layouts should be *templates*, providing instructions for assembling the full page, rather than containing the full page content itself. Currently, the content in `default.html` is in fact only the content for `index.html`. If we want to use `default.html` as a template for additional pages, we need some way to insert the content specific to those pages.

The way to do this in Jekyll is to replace the content in the template with a special Liquid tag that inserts the content dynamically, so that it can be different for each page. The tag itself looks like this:

```
{{ content }}
```

This tag tells Jekyll to insert into the layout file whatever content is on the page that a user is loading (such as the Home page `index.html`). With this new tag in place, we can make as

many additional pages as we want, and each will have its own content that will get built into a final page. All those new pages will have the same header, footer, and any other structure (thereby adhering to the DRY principle (Box 1.3)) that we want all pages to have.

To get this to work, we'll start by refactoring the index page, moving the index-specific content from `default.html` to `index.html`, and then dynamically insert that content in the default template. The first step is to use your text editor to cut the content in `default.html` between the header and footer includes and replace it with the Liquid tag from above. The result should appear as in shown Listing 6.1.

Listing 6.1: *Replacing the content with a Liquid tag.*

_layouts/default.html

```
<!DOCTYPE html>
<html>
  {% include head.html %}
  <body>
    {% include header.html %}

    {{ content }}

    {% include footer.html %}
  </body>
</html>
```

The second step is to paste the content you just cut out of `default.html` into `index.html` (while preserving the Jekyll frontmatter from Listing 5.1). The entire `index.html` should look like Listing 6.2. Because this was a refactoring, the result should appear unchanged (Figure 6.2).

Listing 6.2: *Putting the index content where it belongs.*

index.html

```
---
layout: default
---

<div class="full-hero hero-home">
  <h1>I'm an h1</h1>
  <ul class="social-list">
    <li>
      <a href="http://example.com/" class="social-link">Fb</a>
    </li>
    <li>
      <a href="http://example.com/" class="social-link">Tw</a>
    </li>
    <li>
      <a href="http://example.com/" class="social-link">Gh</a>
    </li>
  </ul>
</div>
<h2>THE FOUNDERS</h2>
<p>
  Learn Enough to Be Dangerous was founded in 2015 by Michael Hartl, Lee Donahoe,
  and Nick Merwin. We believe that the kind of technical sophistication taught by
  the Learn Enough tutorials can benefit at least a billion people, and probably
  more.
</p>
<p>Test paragraph</p>
<div class="bio-wrapper">
  <div class="bio-box">
    <img src="http://placekitten.com/g/400/400">
```

```
      <h3>Michael Hartl</h3>
      <a href="http://twitter.com/mhartl" class="social-link">Tw</a>
      <div class="bio-copy">
        <p>
          Known for his dazzling charm, rapier wit, and unrivaled humility,
          Michael is the creator of the
          <a href="http://railstutorial.org/">Ruby on Rails
          Tutorial</a> and principal author of the
          <a href="https://learnenough.com/">
          Learn Enough to Be Dangerous</a> introductory sequence.
        </p>

        <p>
          Michael is also notorious as the founder of
          <a href="http://tauday.com/">Tau Day</a> and author of
          <a href="http://tauday.com/tau-manifesto"><em>The Tau
          Manifesto</em></a>, but rumors that he's secretly a supervillain
          are slightly exaggerated.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Lee Donahoe</h3>
      <a href="https://twitter.com/leedonahoe" class="social-link">Tw</a>
      <div class="bio-copy">
        <p>
          When he's not literally swimming with sharks or hunting powder
          stashes on his snowboard, you can find Lee in front of his computer
          designing interfaces, doing front-end development, or writing some of
          the interface-related Learn Enough tutorials.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Nick Merwin</h3>
      <a href="https://twitter.com/nickmerwin" class="social-link">Tw</a>
      <div class="bio-copy">
        <p>
          You may have seen him shredding guitar live with Capital Cities on
          Jimmy Kimmel, Conan, or The Ellen Show, but rest assured Nick is a
          true nerd at heart. He's just as happy shredding well-spec'd lines
          of code from a tour bus as he is from his kitchen table.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>??</h3>
      <p>
        The Future
      </p>
    </div>
  </div>
  <p>
    Learn Enough to Be Dangerous is a leader in the movement to teach the
    world <em>technical sophistication</em>, which includes both "hard
    skills" like coding, command lines, and version control, and "soft
     skills" like guessing keyboard shortcuts, Googling error messages, and
    knowing when to just reboot the darn thing.
  </p>
  <p>
    We believe there are <strong>at least a billion people</strong> who can
    benefit from learning technical sophistication, probably more. To join
    our movement,
    <a href="https://learnenough.com/#email_list">sign up for our official
    email list</a> now.
  </p>
  <h3>Background</h3>
  <p>
```

```
  Learn Enough to Be Dangerous is an outgrowth of the
  <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a> and the
  <a href="http://www.softcover.io/">Softcover publishing platform</a>.
  This page is part of the sample site for
  <a href="https://learnenough.com/css-tutorial"><em>Learn Enough CSS and
  Layout to Be Dangerous</em></a>, which teaches the basics of
  <strong>C</strong>ascading <strong>S</strong>tyle
  <strong>S</strong>heets, the language that allows web pages to be styled.
  Other related tutorials can be found at
  <a href="https://learnenough.com/">learnenough.com</a>.
</p>
```

Figure 6.2: The unchanged index page.

The way this works is that Jekyll automatically collects the content in `index.html` (i.e., everything in Listing 6.2 other than the frontmatter) into a special variable (Box 6.1)[1] called `content`, which then gets inserted by the Liquid command in Listing 6.1 into the template specified by `layout: default`.

Box 6.1. What is a variable?

If you've never programmed a computer before, you may be unfamiliar with the term *variable*, which is an essential idea in computer science. You can think of a variable as a named box that can hold different (or "variable") content.

As a concrete analogy, consider the labeled boxes that many elementary schools provide for students to store clothing, books, backpacks, etc. (Figure 6.3). The label for the box is the variable name, and the contents of the box is the variable content. Box-plus-label is the variable itself.

In the context of a Jekyll template, the content on the page (such as `index.html`) is automatically collected into a variable called `content`, which can be inserted into the

template using the special command `{{ content }}` as described above (Listing 6.1).

Figure 6.3: A concrete manifestation of computer variables.

Note that `{{ content }}` is a generic instruction to Jekyll to insert the value of the `content` variable, which in this context is the content from whatever page the user has visited (e.g., `index.html`). It's kind of a wildcard, as opposed to Liquid tags like `{% include footer.html %}` (Section 5.6), which include only a specific file's contents. We'll be talking about variables more in Section 8.1.

We'll put our new fully flexible layout template to good use in Section 6.4, and again in Section 8.1. First, though, we should probably make our homepage look a little less boring.

6.1.1 Exercises

1. Make a new file called `test.html` in your root directory, specify the default layout in the frontmatter, and for content add something like "For the fourth, and I hope final time, hello, world". Visit http://localhost:4000/test.html in your browser to verify that it worked.

2. Create a new file in your `_layouts` directory, and add in only a `{{ content }}` tag. Then change the `index.html` frontmatter to use that new layout to build the page. Visit http://localhost:4000/ in your browser to see the result.

6.2 There's no place like home

We'll start our homepage styling right at the top with the hero section that we created all the way back in Section cha:values_view. As we mentioned there, you've likely seen a lot of sites with nice big images at the top of the page, and these sections are called a variety of different names—hero sections, billboards, splashes, etc.—but, whatever the name, the goal is the same: to add a big attractive graphical element to the page to set the tone and grab the reader's attention. But our current plain blue hero is a bit of a zero... so we should change that.

As preparation for the hero improvements, we're going to update the markup on our index page a bit. The new `index.html` appears in Listing 6.3. Notice that we've taken the text that was at the bottom of the page and wrapped it inside of new divs with new classes (everything inside the `div` with the class of `.home-callout`), and also moved it to the top of the page. We've also eliminated the final `.bio-box` with the question marks.

You may find it easier to copy and paste from Listing 6.3 rather than rewrite it all yourself, but if you feel like getting your hands dirty feel free to adapt the content on your page so that it matches. The result should look something like Figure 6.4.

Listing 6.3: *Updating the HTML on the Home page.*

index.html

```
---
layout: default
---

<div class="full-hero hero-home">
  <h1>I'm an h1</h1>
  <ul class="social-list">
    <li>
      <a href="http://example.com/" class="social-link">Fb</a>
    </li>
    <li>
      <a href="http://example.com/" class="social-link">Tw</a>
    </li>
    <li>
      <a href="http://example.com/" class="social-link">Gh</a>
    </li>
  </ul>
</div>

<div class="home-callout">
  <h1 class="callout-title">The Learn Enough Story</h1>
  <div class="callout-copy">
    <p>
      Learn Enough to Be Dangerous is a leader in the movement to teach the world
      <em>technical sophistication</em>, which includes both "hard skills" like
      coding, command lines, and version control, and "soft skills" like guessing
      keyboard shortcuts, Googling error messages, and knowing when to just
      reboot the darn thing.
    </p>
    <p>
      We believe there are <strong>at least a billion people</strong> who can
      benefit from learning technical sophistication, probably more. To join our
      movement, <a href="https://learnenough.com/#email_list">sign up for our
      official email list</a> now.
    </p>
    <h3>Background</h3>
    <p>
      Learn Enough to Be Dangerous is an outgrowth of the
```

```
        <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a> and the
        <a href="http://www.softcover.io/">Softcover publishing platform</a>. This
        page is part of the sample site for <a
        href="https://learnenough.com/css-tutorial"><em>Learn Enough CSS and Layout
        to Be Dangerous</em></a>, which teaches the basicics of
        <strong>C</strong>ascading <strong>S</strong>tyle
        <strong>S</strong>heets, the language that
        allows web pages to be styled. Other related tutorials can be found at
        <a href="https://learnenough.com/">learnenough.com</a>.
      </p>
    </div>
  </div>

  <div class="home-section">
    <h2>THE FOUNDERS</h2>
    <p>
      Learn Enough to Be Dangerous was founded in 2015 by Michael Hartl, Lee
      Donahoe, and Nick Merwin. We believe that the kind of technical
      sophistication taught by the Learn Enough tutorials can benefit at least a
      billion people, and probably more.
    </p>

    <div class="bio-wrapper">
      <div class="bio-box">
        <img src="http://placekitten.com/g/400/400">
        <h3>Michael Hartl</h3>
        <a href="http://twitter.com/mhartl" class="social-link">
          Tw
        </a>
        <div class="bio-copy">
          <p>
            Known for his dazzling charm, rapier wit, and unrivaled humility,
            Michael is the creator of the
            <a href="http://railstutorial.org/">Ruby on Rails
            Tutorial</a> and principal author of the
            <a href="https://learnenough.com/"> Learn Enough to Be Dangerous</a>
            introductory sequence.
          </p>
          <p>
            Michael is also notorious as the founder of
            <a href="http://tauday.com/">Tau Day</a> and author of
            <a href="http://tauday.com/tau-manifesto"><em>The Tau
            Manifesto</em></a>, but rumors that he's secretly a supervillain are
            slightly exaggerated.
          </p>
        </div>
      </div>
      <div class="bio-box">
        <img src="http://placekitten.com/g/400/400">
        <h3>Lee Donahoe</h3>
        <a href="https://twitter.com/leedonahoe" class="social-link">
          Tw
        </a>
        <div class="bio-copy">
          <p>
            When he's not literally swimming with sharks or hunting powder stashes
            on his snowboard, you can find Lee in front of his computer designing
            interfaces, doing front-end development, or writing some of the
            interface-related Learn Enough tutorials.
          </p>
        </div>
      </div>
      <div class="bio-box">
        <img src="http://placekitten.com/g/400/400">
        <h3>Nick Merwin</h3>
        <a href="https://twitter.com/nickmerwin" class="social-link">
          Tw
        </a>
        <div class="bio-copy">
          <p>
            You may have seen him shredding guitar live with Capital Cities on
```

```
        Jimmy Kimmel, Conan, or The Ellen Show, but rest assured Nick is a true
        nerd at heart. He's just as happy shredding well-spec'd lines of code
        from a tour bus as he is from his kitchen table.
      </p>
    </div>
  </div>
</div>
</div>
```

Figure 6.4: An updated index page.

To get started on the hero update, let's first increase the size of the `.full-hero` class that we styled in Section cha:values_view to make it the size of the entire browser window. To do this, we'll set the `height` to 100% of the viewport height using `height: 100vh` (Section cha:values_view):

```
.full-hero {
  height: 100vh;
}
```

We'll also add in a background image (an extremely *dangerous* shark) using the `background-image` attribute, which takes an absolute path (with a leading slash `/` in `/images/shark.jpg`) as follows:

```
.hero-home {
  background-image: url(/images/shark.jpg);
}
```

(We'll download `shark.jpg` in a moment.)

Putting these elements together, and adding a few extra rules for the `.full-hero` class, gives us Listing 6.4.

Listing 6.4: *New styles for the hero section.*

css/main.css

```
.
.
.
/* FOOTER STYLES */
.
.
.
/* HERO STYLES */
.full-hero {
  background-color: #c7dbfc;
  box-sizing: border-box;
  height: 100vh;
  padding-top: 10vh;
}
.hero-home {
  background-image: url(/images/shark.jpg);
}
.
.
.
```

Note in Listing 6.4 the placement of the hero styles below site-wide styles like those for the header and footer. The reasoning is that the header and footer sections of CSS tend to stay relatively unchanged through updates, whereas you often find yourself making additions to the CSS that deal with the site content, so it makes sense to me to have things that change less often at the top of a file, and add new stuff to the bottom.

Note also that we've added in a `border-box` style to make sure that the padding we'll add doesn't affect the overall element height (Section 4.2), and we've added padding to the top so that content isn't covered up by the fixed-position header.

To get the hero image, use `curl` to download `shark.jpg` to your local disk:[2]

```
$ curl -o images/shark.jpg -OL cdn.learnenough.com/le-css/shark.jpg
```

Save and refresh and... hmm... Figure 6.5 doesn't look quite right.

Figure 6.5: Our hero image is looking a little less than heroic.

What happened? Well, when the browser loads an image as a background, it isn't any different than when you put an image on a page using an HTML `img` tag—the image is displayed at its full dimensions. We've changed the size of images in this tutorial by giving a width or height to the `img` element in CSS (e.g., Listing 5.33), but the `height` and `width` properties don't work for background styles. To resize a background image to fit a container, we need to use the `background-size` style, as shown in Listing 6.5.

Listing 6.5: *Adding the style to resize the background image.*

css/main.css

```
.
.
.
.full-hero {
  background-color: #c7dbfc;
  background-size: cover;
  box-sizing: border-box;
  height: 100vh;
  padding-top: 10vh;
}
.hero-home {
  background-image: url(/images/shark.jpg);
}
.
.
.
```

Setting the `background-size` property to `cover` means that the image will be resized so that the entire `.full-hero` container is covered by the image, even if that means cutting off parts of the image (Figure 6.6).

Figure 6.6: Resizing the image just enough to fully cover the background.

You can also use the value `contain` and the background image will be resized so that the entire picture will always be resized to entirely fit inside the object, but that can lead to a situation where the image doesn't fully cover the background (Figure 6.7).

Figure 6.7: Using `background-size: contain` can leave part of the background showing.

Now reset back to using `background: cover`. If you resize your browser window to make it really narrow, you'll see that the image is dynamically resized, but you'll also likely notice that the resizing is done with the anchor point in the top left. Unfortunately, our shark is in the center of the image. When the window gets narrow, it starts getting cut off again (Figure 6.8).

Figure 6.8: By default, the resizing is anchored at the top left of the image.

That's not always ideal, but luckily we can change the anchor point for the background image by using the `background-position` property (Listing 6.6).

Listing 6.6: *Changing the anchor point for where the browser resizes the image.*

css/main.css

```
.
.
.
.full-hero {
  background-color: #c7dbfc;
  background-size: cover;
  box-sizing: border-box;
  height: 100vh;
  padding-top: 10vh;
}
```

```
.hero-home {
  background-image: url(/images/shark.jpg);
  background-position: center top;
}
.
.
.
```

Now the shark is better-positioned inside the element (Figure 6.9).

Figure 6.9: Resizing the picture from an anchor point at the center top.

The `background-position` style allows us to position a background with specific values like pixels or percentages, or we can use generic terms like `center`, `top`, `left`, etc. The first value controls the x-axis and the second the y-axis. So if we gave the background the style `background-position: center center` then any resizing would always keep the middle of the image in the middle of the container `div`.

Next let's make it so that the text, and the social links, are always vertically aligned in the hero using the positioning tricks we learned in Section 5.8. We could absolutely position that content, or we could just set the wrapper's position property to `relative`, and then nudge things around.

First, let's move the `h1` and the `.social-links` list into a new wrapper element, with a class of `.hero-content`, so that we are only adding positional styling to a single HTML element that

will then contain all the rest of the content in the hero (Listing 6.7). Note that we've also updated the `h1` to read CODE DANGEROUSLY.

Listing 6.7: *Wrapping the hero's content inside a new* `div`.

index.html

```
.
.
.
<div class="full-hero hero-home">
  <div class="hero-content">
    <h1>CODE DANGEROUSLY</h1>
    <ul class="social-list">
      <li>
        <a href="http://example.com/" class="social-link">Fb</a>
      </li>
      <li>
        <a href="http://example.com/" class="social-link">Tw</a>
      </li>
      <li>
        <a href="http://example.com/" class="social-link">Gh</a>
      </li>
    </ul>
  </div>
</div>
.
.
.
```

If you wanted to absolutely position the `.hero-content`, then you would need to remember to add a `position: relative` to the `.full-hero` class so that it could act as the container for absolutely positioned elements inside of it. Since there isn't anything else in the hero, we don't need to worry about taking the `.hero-content` entirely out of the document flow. Instead, we can just set the `.hero-content` to `position: relative` and then nudge it around. At the same time, we'll do some styling for the `.social-links` and the `h1`.

Since the `.hero-content` wrapper is already going to be 100% of the width (it's just a normal block element), we don't need to worry about messing with horizontal centering. Just aligning the content using `text-align: center` will be good enough—we only need to worry about vertical positioning, which we can do with the `top: 50%` and `translate(0, -50%)` (both of which you've seen in Section 5.8):

```
top: 50%;
transform: translate(0, -50%);
```

We'll then style the hero's `h1` with a partially transparent color, a big font size, and a bottom margin:

```
color: rgba(255, 255, 255, 0.8);
font-size: 7vw;
margin-bottom: 0.25em;
```

All together, our existing and new hero styling is shown in Listing 6.8.

Listing 6.8: *Positioning the hero content to be vertically centered.*

css/main.css

```
.
.
.
.hero-content {
  color: #fff;
  position: relative;
  text-align: center;
  text-transform: uppercase;
  top: 50%;
  transform: translate(0, -50%);
}
.hero-content h1 {
  color: rgba(255, 255, 255, 0.8);
  font-size: 7vw;
  margin-bottom: 0.25em;
}
.hero-content .social-link {
  background-color: rgba(255, 255, 255, 0.8);
  color: #557c83;
}
.hero-content .social-link:hover {
  background-color: #000;
  color: #fff;
}
.
.
.
```

The hero now looks nicely styled (Figure 6.10).

Figure 6.10: This hero section is starting to look... DANGEROUS(LY GOOD)!

6.2.1 Exercises

1. The background-size property isn't just limited to things like `cover` and `contain`, try setting the width value to a hard number like `300px` and the height to `auto`. What

would that shorthand value be?

2. Now that you have a smaller background image, you'll have noticed that the browser repeats it to fill the element's background. We can stop that though. Try adding a style of `background-repeat: no-repeat` to the `.full-hero` element.

3. Similar to background-size, the background-position property can take number values. Change the positioning to `30% center`.

6.3 More advanced selectors

The hero from Section 6.2 is looking great, but there's a missing detail we'd like to add. When people visit our site, there's a risk that they won't realize there is content lower down on the page. What if we added a little downward-pointing arrow at the bottom of the hero to give them a hint they should scroll?

One way we could do it would be to add a new `div` in the hero, give it a class name, and give that some styles, maybe use an image for the arrow, etc.—but that would be adding extra stuff to the code that would just clutter things up.

There is, however, another advanced CSS technique we can use to add stuff onto the page without adding anything to the HTML...

In this section, we'll see how to use the advanced *pseudo-element* to add a downward-pointing arrow to our hero section (Section 6.2) without using any images and with a minimum of fuss.

6.3.1 The :before/:after pseudo-element

We introduced the *pseudo-class* when we were styling links in Section 5.7.1, and if you remember it is kind of like a pretend CSS class that applies only when a user interacts with an element in a certain way (such as hovering over it with the mouse or visiting a link).

All HTML elements also have two pretend pseudo-elements, called `:before` and `:after`. Even though this might sound super-technical and weird, surprisingly you've already used pseudo elements, once before when we played around with clearing floats (Section 4.3.1), and also the very first time that you ever made a bullet list (as in the Lists section of *Learn Enough HTML to Be Dangerous*). You didn't know it, but when you created that bullet list, the individual bullet points were actually placed on the page by the browser using `:before` pseudo-elements.

The "before" and "after" in the pseudo-element names just refer to where the corresponding element will show up. So, if you have a line of text, `:before` will show up at the beginning, and `:after` will show up at the end (Figure 6.11)... shocking, right?

Figure 6.11: The position of the `:before` *and* `:after` *pseudo-elements.*

Because these pseudo-elements were originally created to modify text, they are by default `inline` elements, and since the default is that they are empty, the result is that in their default rendering they take up no space. Here's the cool thing, though: they work just as if you added in a `span` into a line of text, gave it a class, and then styled it. You can do all the same sorts of things that you'd do to any element on the page—change the display type, move it around, change how it looks—whatever your heart desires! They are perfect for adding little details to elements on a site without needing to complicate your code with extra HTML tags and classes.

We'll do a quick example just to illustrate the concept, and then actually implement the down arrow mentioned above. Start by adding the styles from Listing 6.9 to your CSS.

Listing 6.9: *A pseudo-element example.*

css/main.css

```
.
.
.
/* HERO STYLES */
.
.
.
.hero-content h1:before {
  color: blue;
  content: "B";
}
.hero-content h1:after {
  color: red;
  content: "A";
}
.
.
.
```

When you save and refresh your browser, you'll see that a B has appeared at the beginning of the title where the `:before` element lives, and there is an A where the `:after` can be found (Figure 6.12).

Figure 6.12: Demonstrating how pseudo-elements fit into ordinary elements.

The CSS property `content` allows you to set what the browser should put into either the `:before` or `:after` location, but the value can only be text (don't try and cram HTML in there!). So even though you didn't add any HTML, you now have new text content on the page.

OK, let's delete those two test styles from the CSS, and add in something useful that applies the `:after` pseudo-element. It's a good practice to place any pseudo-element CSS immediately below the parent style in the CSS file, so we'll locate the `.full-hero:after` rules immediately after the `.full-hero` section. The result, shown in Listing 6.10, includes a Unicode vertical right angle bracket, which we suggest you copy and paste, as your keyboard might not be able to produce it. (We aren't going to get into Unicode any further in this tutorial, but if you flip through a site like Unicode character table you'll be surprised at how many elements that would be useful in web design are already part of the Unicode character set.)

Listing 6.10: *Apply a style to the* `:after` *pseudo-element.*

css/main.css

```
.
.
.
/* HERO STYLES */
.full-hero {
  background-color: #c7dbfc;
  background-size: cover;
  box-sizing: border-box;
  height: 100vh;
  padding-top: 10vh;
}

.full-hero:after {
```

```
  bottom: 2vh;
  color: #fff;
  content: "﹀";
  font-size: 36px;
  left: 50%;
  position: absolute;
  transform: translate(-50%, 0);
}
.
.
.
```

With the CSS in Listing 6.10, you should have a white down arrow at the bottom of the hero (Figure 6.13).

Figure 6.13: A wild down-pointing arrow has appeared!

6.3.2 :before/:after CSS triangle

Want to see another neat trick? If you ever want to make a triangle with just CSS, you can do so easily thanks to a quirk with the way that borders are drawn in browsers. Let's change that `:after` that we just created in Listing 6.10 to illustrate the concept (Listing 6.11).

Listing 6.11: *Adding a multi-colored thick border.*

`css/main.css`

```
.
.
.
.full-hero:after {
  border: 10px solid;
  border-color: #000 red purple blue;
  bottom: 2vh;
  content: "";
  height: 0;
  left: 50%;
```

```
  position: absolute;
  transform: translate(-50%, 0);
  width: 0;
}
.
.
.
```

When you save and refresh the browser, you'll see that the downward-pointing caret has been replaced by a colored square that is divided into 4 triangular sections (Figure 6.14).

Figure 6.14: A zero-height and zero-width element with a thick border.

It turns out that browsers draw the line between, say, the top and right border at a 45° angle.[3] Normally, you don't notice this because your borders are probably only 1px, and the same color all the way around. But, if you make an element with zero height and zero width, and then give it a big border that has different colors on all sides, you'll find the rendering oddity shown in Figure 6.14. (We have no idea who the first person was that figured this out or why they even were trying it.)

So how do we use this to our advantage?

One of the acceptable "colors" for a border is `transparent` (alternatively, you could use `rgba(0, 0, 0, 0)`). So, recalling from Section cha:box_border that `border-color` uses a margin-style shorthand, if we style the element as in Listing 6.12...

Listing 6.12: *Finishing the styling for the downward-pointing triangle.*

css/main.css

```
.
.
.
.full-hero:after {
  bottom: 2vh;
  border: 10px solid;
  border-color: #fff transparent transparent;
  content: "";
  height: 0;
  left: 50%;
```

```
  position: absolute;
  transform: translate(-50%, 0);
  width: 0;
}
.
.
.
```

...we'll end up with a downward-pointing triangle (Figure 6.15)!

Figure 6.15: People seriously used to have to make images to do this kind of thing... How barbaric.

Pretty cool, and actually very useful in menus and navigation for showing users what they are currently looking at. You can use the `:before` and `:after` pseudo-elements to do a lot of really cool things without adding unnecessary HTML to your pages—it's perfect for little design elements like this. You just have to deal with the limitation that you can't put extra HTML into the pseudo objects.

Before we leave the homepage, let's give the remaining sections a little bit of structure and style tweaking. One of the things that we've mentioned a number of times is how important it is for things to be modular, and right now on the homepage we have some elements that are inside wrappers, and some that are just loose on the page.

In this sort of situation, it's a good idea to add `div` wrappers (Box 6.2)[4] to wrap content that belongs in a single unit. That way, if you want the different sections on the homepage to be exactly `10vh` apart from each other, the styling is easy. That's why Listing 6.3 wrapped the founders `h2`, the paragraph that follows, and the `.bio-box` es in a `div` with the class `.home-section`, as reviewed in Listing 6.13.

Listing 6.13: *The repackaged content on the homepage.*

index.html

```
.
.
.
<div class="home-callout">
  <h1 class="callout-title">The Learn Enough Story</h1>
  <div class="callout-copy">
    <p>
      Learn Enough to Be Dangerous is a leader in the movement to teach the world
      <em>technical sophistication</em>, which includes both "hard skills" like
      coding, command lines, and version control, and "soft skills" like guessing
      keyboard shortcuts, Googling error messages, and knowing when to just
      reboot the darn thing.
    </p>
    <p>
      We believe there are <strong>at least a billion people</strong> who can
      benefit from learning technical sophistication, probably more. To join our
      movement, <a href="https://learnenough.com/#email_list">sign up for our
      official email list</a> now.
    </p>
    <h3>Background</h3>
    <p>
      Learn Enough to Be Dangerous is an outgrowth of the
      <a href="http://railstutorial.org/">Ruby on Rails Tutorial</a> and the
      <a href="http://www.softcover.io/">Softcover publishing platform</a>. This
      page is part of the sample site for <a
      href="https://learnenough.com/css-tutorial"><em>Learn Enough CSS and Layout
      to Be Dangerous</em></a>, which teaches the basicics of
      <strong>C</strong>ascading <strong>S</strong>tyle
      <strong>S</strong>heets, the language that
      allows web pages to be styled. Other related tutorials can be found at
      <a href="https://learnenough.com/">learnenough.com</a>.
    </p>
  </div>
</div>

<div class="home-section">
  <h2>THE FOUNDERS</h2>
  <p>
    Learn Enough to Be Dangerous was founded in 2015 by Michael Hartl, Lee
    Donahoe, and Nick Merwin. We believe that the kind of technical
    sophistication taught by the Learn Enough tutorials can benefit at least a
    billion people, and probably more.
  </p>

  <div class="bio-wrapper">
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Michael Hartl</h3>
      <a href="http://twitter.com/mhartl" class="social-link">
        Tw
      </a>
      <div class="bio-copy">
        <p>
          Known for his dazzling charm, rapier wit, and unrivaled humility,
          Michael is the creator of the
          <a href="http://railstutorial.org/">Ruby on Rails
          Tutorial</a> and principal author of the
          <a href="https://learnenough.com/">
          Learn Enough to Be Dangerous</a> introductory sequence.
        </p>
        <p>
          Michael is also notorious as the founder of
          <a href="http://tauday.com/">Tau Day</a> and
          author of <a href="http://tauday.com/tau-manifesto"><em>The Tau
          Manifesto</em></a>, but rumors that he's secretly a supervillain are
          slightly exaggerated.
        </p>
```

```
      </div>
    </div>
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Lee Donahoe</h3>
      <a href="https://twitter.com/leedonahoe" class="social-link">
        Tw
      </a>
      <div class="bio-copy">
        <p>
          When he's not literally swimming with sharks or hunting powder stashes
          on his snowboard, you can find Lee in front of his computer designing
          interfaces, doing front-end development, or writing some of the
          interface-related Learn Enough tutorials.
        </p>
      </div>
    </div>
    <div class="bio-box">
      <img src="http://placekitten.com/g/400/400">
      <h3>Nick Merwin</h3>
      <a href="https://twitter.com/nickmerwin" class="social-link">
        Tw
      </a>
      <div class="bio-copy">
        <p>
          You may have seen him shredding guitar live with Capital Cities on
          Jimmy Kimmel, Conan, or The Ellen Show, but rest assured Nick is a true
          nerd at heart. He's just as happy shredding well-spec'd lines of code
          from a tour bus as he is from his kitchen table.
        </p>
      </div>
    </div>
  </div>
</div>
```

Box 6.2. Style Note: What's the deal with all these wrappers?

So, what's the deal with all these wrappers?

Well, we could just add a `.home-section` class to an element like the `.bio-wrapper` and style it up, but then we'd lose the ability to pull that entire `.bio-wrapper` out of the homepage and seamlessly drop it onto another part of the site (like on an About page), since it would potentially have an extra class on it that pertains only to the homepage.

We could make sure that the styling attached to that homepage class wouldn't affect the layout of the content if you put it on another page, but then we could end up having really

complicated selectors and styling that is just undoing a bunch of properties, and then redoing new styles. To see what we mean, let's look at the Apple homepage for the Mac. Scrolling down the page, you'll see different sections that have similar styling for height, background, and margins/padding. Setting things up so that you have common containers for content makes your job easier, and these generic wrappers that have styling like height, margins, or padding makes it easy to flexibly add new content in the future.

Let's also add some styles to these new sections and to some of the content. There are a lot of changes, so feel free to save and refresh as we're going along. We'll have a screenshot showing the cumulative changes at the end.

For the `.home-section` s themselves, we are going to size them to be `90vw` and also give them an upper limit, and then use the `margin: auto` trick from Section cha:box_margin-auto (plus some top and bottom margin to keep them away from other content) to keep the sections in the center of the page:

```
margin: 6rem auto;
max-width: 980px;
width: 90vw;
```

We are going to style the `h2` so that it is centered, and has some distance from content below it:

```
margin-bottom: 1.5rem;
text-align: center;
```

We'll also style the callouts to give them background and text color, and nice padding:

```
background-color: #000;
color: #fff;
padding: 7vh 0;
```

And lastly we'll do some text styling in that section, first for the title to make it nice and big:

```
font-size: 5.75vw;
text-align: right;
text-transform: uppercase;
```

Then for the regular text we'll set the font-size to be a little smaller than normal to prevent it from being too overwhelming on the page:

```
font-size: 0.8rem;
```

All together that looks like Listing 6.14.

Listing 6.14: *Styling the home section and callout.*

css/main.css

```
.
.
.
/* SOCIAL STYLES */
.
.
.
```

```
/* HOMEPAGE STYLES */
.home-section {
  margin: 6rem auto;
  max-width: 980px;
  width: 90vw;
}
.home-section h2 {
  margin-bottom: 1.5rem;
  text-align: center;
}
.home-callout {
  background-color: #000;
  color: #fff;
  padding: 7vh 0;
}
.callout-title {
  font-size: 5.75vw;
  text-align: right;
  text-transform: uppercase;
}
.callout-copy {
  font-size: 0.8rem;
}
.
.
.
```

Now we are going to attack the `.bio-box` es. Let's get rid of the borders, resize the width of the boxes to fully fill the page now that there are only three, change the margins and padding to give us nicer spacing between the elements, tweak the header styles so that the names look a little nicer, and then finally center the social links after giving them a little space on top to separate them from the images. We can also get rid of the link color styling for the bios, the line-height style on the copy in the bio text (we'll change it so that all paragraphs on the site have reasonable line height), and then also style the text a little with some new sizes. The result appears in Listing 6.15.

Listing 6.15: *Restyling the* `.bio-boxes`.

css/main.css

```
.
.
.
.bio-box {
  box-sizing: border-box;
  float: left;
  font-size: 1rem;
  margin: 6rem 0 0;
  padding: 0 3%;
  width: 33%;
}
.
.
.
.bio-box h3 {
  color: #fff;
  font-size: 1.5em;
  margin: -40px 0 1em;
  text-align: center;
  text-transform: uppercase;
}
.
.
.
.bio-box .social-link {
```

```
  display: block;
  margin: 2em auto 1em;
}
.bio-copy {
  font-size: 0.75em;
}
```

Up at the top of the CSS file, we'll address the line-height issue with a global line-height style for paragraphs. This gives text a little more breathing room, and while we are there, we'll also change the default link color to something a little more pleasant than the bright red (Listing 6.16).

Listing 6.16: *Adding in some new global styles.*

```
 css/main.css

/* GLOBAL STYLES */
.
.
.
h1 {
  font-size: 7vw;
  margin-top: 0;
}
h2 {
  font-size: 2em;
}
a {
  color: #6397b5;
}
p {
  line-height: 1.5;
}
h2 ~ p {
  .
  .
  .
}
.
.
.
```

Overall, a more pleasing design (Figure 6.16)!

Figure 6.16: The improved bio section.

6.3.3 Exercises

1. One common mistake when dealing with `:before` and `:after` is forgetting to define the `content` property. Try deleting that from the `.full-hero:after` styling. What happens?

2. Another common mistake is forgetting that these pseudo-elements are by default inline elements. What happens when you delete `position: absolute` from `.full-hero:after`? Before adding it back in, trying setting the element to `display: block`.

6.4 Other pages, other folders

Now that we've polished up the home page a bit, let's add a second page to see how to re-use the layout defined in Listing 6.1. To do that, we'll need to add a new file into our project.

The easiest way to add new pages to a Jekyll site is to simply put other HTML files in the root directory, and then link to them the way that we did in *Learn Enough HTML to Be Dangerous*—for example, like this (Figure 6.17):

```
http://localhost:4000/pagename.html
```

Figure 6.17: This is kind of an ugly URL for a page.

That URL is kind of ugly though, and these days most modern sites have pages with nice-looking addresses like http://sitename.com/pagename, without the `.html` at the end (Figure 6.18). It seems like a little thing, but URLs are UI, and people really do notice details like that on sites. If something doesn't look right, they might think that your site isn't trustworthy or professional.

Figure 6.18: This kind of URL looks much cleaner.

Lucky for us, Jekyll gives developers a method to approximate that kind of "pretty URL", so let's try it out.[5]

The trick is that you can add a directory into your project, and then create a file in the directory called `index.html`. As long as the directory has a name that conforms to URL standards, you can link and visit that index page by just using `/directoryname`.

To see how this works, create a new directory in the project called `gallery`, and create a new `index.html` file inside that directory:

```
$ mkdir gallery
$ touch gallery/index.html
```

(Repeating the `index.html` filename may seem confusing, but this convention of using `index.html` for the main file in a directory is as old as the Web.)

Once the file is created, add in the frontmatter that tells Jekyll which layout to use (and a little bit of text), and save your work (Listing 6.17).

Listing 6.17: *Adding an index page inside a new directory.*

gallery/index.html

```
---
layout: default
---

I'm a 3 col page!
```

Your directories and files should now look something like Figure 6.19.

Figure 6.19: The new additions in the project directory.

Now go to http://localhost:4000/gallery. As promised in Section 6.1, Jekyll has inserted the content of the relevant page (in this case, Listing 6.17) into the default template. As a result, the pretty URL works, but unfortunately the text doesn't display (Figure 6.20), even though (as you can verify using a web inspector) the text is present in the page's source.

Figure 6.20: This new gallery page doesn't even show the text.

In looking at our new page, we have just run into one of the age-old problems in creating websites… a page is only as big as the content you added to it. All that white stuff at the bottom of Figure 6.20 is actually not even part of your page—it's just the default background the browser is adding to fill the window (Figure 6.21).

Figure 6.21: Your text is somewhere… up there.

In this case, the content (one line of text) is getting covered up by the fixed-position header, and then the footer is drawn right after that. Ideally, you want the footer at least at the bottom of the browser so that you don't see this default page background, but there isn't enough content to push it down.

Over the years there have been a variety of different methods for solving this problem, some more complex than others, most requiring a bunch of wrapper divs, plus `min-height` set on some things, `height: 100%` on others, and a dash of negative margin (read: complicated). But, there is a new styling method in town that allows developers to simply and efficiently solve this problem: the CSS flexbox.

As we'll see in Chapter 7, not only will the CSS flexbox solve our text-clearing problem, it will also allow us to easily make multi-column page layouts (thereby fulfilling the promise implicit in the gallery text "I'm a 3 col page!"). The result will be the beginnings of a simple photo gallery, which will come to full fruition in *Learn Enough JavaScript to Be Dangerous*.

6.4.1 Exercises

1. Create another page called `test.html` in the `gallery` folder. What happens when you try to go to http://localhost:4000/gallery/test? What about http://localhost:4000/gallery/test.html?

2. How would you create another nicely formatted URL that points to a page inside the gallery folder?

Chapter 7
Specialty page layouts with flexbox

As you might guess from the name, CSS *flexbox* is a flexible box model for laying out content on the Web. Flexbox, which is a relatively new addition to CSS, makes it possible to style sections of a website to allow the sections to fill a space, while also still being able to adapt to the content—a combination that was often difficult to do in the pre-flexbox era.

For example, suppose we had three columns with different amounts of content inside, and we wanted the columns all to be the same height as the *longest* one. A diagram of this scenario for three columns *without* flexbox appears in Figure 7.1.

Figure 7.1: Columns with different lengths of content, and hence different heights.

Back in the days before flexbox, it was surprisingly hard to add styles to transform the content in Figure 7.1 to look the content in Figure 7.2. It required doing things like using JavaScript to check the height of elements every time the window changed size, or using HTML tables for layout (which you should never do (Box 7.1)).

Figure 7.2: How we want the columns to look—a surprisingly difficult problem.

Now that we have flexbox, this type of problem has an easy solution! In this section, we'll apply simplified flexbox rules to fix the layout problem left at the end of Section 6.4 (Figure 6.21), and we'll also take the opportunity to polish up the callout section on the homepage (Section 7.2). With these examples in hand, we'll then take a look at more advanced flexbox features, including a powerful shorthand notation (Section 7.3). Finally, we'll apply these more advanced features to create a *three-column* layout for the gallery introduced in Section 6.4.

Box 7.1. Style Note: Never use tables for layout.

Back in the bad old days of the early Internet, when there was no CSS, the only way to create page layouts was to use tables inside tables (inside tables), and then put content in the table cells. This was a terrible abuse of the `table` tag, which is designed for organizing and displaying tabular data (like spreadsheets), but unfortunately there was no other way to arrange things on a page.

As time went on, developers were given new tools, elements, and styles for creating layouts, but there were still gaps in the toolset, which made it difficult to create layouts like the multiple columns shown in Figure 7.2. So developers continued to use tables to make their designs work (for certain values of "work").

The problem is that, beyond the semantic problem of using an element meant for data to create a layout, the table layout is incredibly strict in how it is displayed. Once you have things arranged in table rows and table cells, that's how the browser has to show them. So if you resize a window with a table, all the content gets squished, and there is no way to restack it, as illustrated in Figure 7.3.

Luckily, with CSS and flexbox, we can now design flexible and robust styles without resorting to tables. Huzzah!

Figure 7.3: Seriously, never use tables for layouts—only for tabular data.

7.1 Having content fill a container

We'll start with an overview of the anatomy of a flexbox. There are two main aspects, the *flex container* and the *flex items*, as illustrated in Figure 7.4, where the flex container is the HTML element with the CSS property `display: flex` set, and each flex item is a child element that has some value of the CSS `flex:` property set. Essentially, the flex container encloses the items, which can be aligned, stretched, shrunk, etc., by various flexbox style rules. (If you are looking at the fine print in Figure 7.4, don't worry right now about the style declaration `flex: 1 1 0` under the flex items; we'll cover it in detail in Section 7.3.)

Figure 7.4: Making flexible containers with child items that fill the available space.

As our first practical application of flexbox items and containers, we'll solve a problem we had at the end of Section 6.4 (Figure 6.21). Recall that the text on the new gallery page wasn't visible even though it was present in the page's source, and the footer wasn't at the bottom of the window, but rather was crushed up against the content. We'll fix both issues by using a flexbox layout that will let us push the footer to the bottom of the screen by vertically filling the available space (Figure 7.5), and also will let us add in padding to move the page content down so that it gets out from under the header.

Figure 7.5: Stretching the elements vertically with flexbox (note the `flex-direction`).

In order to apply flexbox to the default template, we'll need to designate both a flex container and some number of flex items (Figure 7.4). Although we could add a new `div` wrapper to use as a flex container, we'll take a simpler approach and simply use the layout's `body` tag. Using a default element like the `body` tag as part of our layout works for this situation because we want the footer to be at the bottom of the window, and/or below the content, on all of the pages across the site.

When we last saw the default site template, the body of the page had two main defined sections, the header and footer, as well as a content tag that loads page content into the layout (Listing 7.1).

Listing 7.1: *The current state of the default template.*

index/default.html

```
<!DOCTYPE html>
<html>
  {% include head.html %}
  <body>
    {% include header.html %}

    {{ content }}

    {% include footer.html %}
  </body>
</html>
```

In order to apply the ideas in Figure 7.5, we need to make some element on the page our flexbox item for the content, and we'll do this using a `div` tag with class `content-container`. (This content "container" is for now a flexbox *item*, but see Section 7.4 below, where it is also a flexbox container—flexboxes within flexboxes!) The updated default layout appears in Listing 7.2.

Listing 7.2: *Wrapping the site content in a new container.*

_layouts/default.html

```
<!DOCTYPE html>
<html>
  {% include head.html %}
  <body>
    {% include header.html %}

    <div class="content-container">
      {{ content }}
    </div>

    {% include footer.html %}
  </body>
</html>
```

Recall from Listing 5.14 and Listing 5.37 that the header and footer are wrapped in the semantic tags `header` and `footer` (which are effectively `div` s). Both elements could act as "flex items" for the purposes of our flexbox layout, but we aren't going to set them to flex as we want them to stay the size that they are—we want only the content area to expand and contract to fill empty space.

We set the `header` to be `position: fixed` , which means it will be taken out of the page flow, and won't be affected by our flexbox setup (Section 5.8). Likewise, we aren't going to target the `footer` with a `flex:` property; instead, it will remain a regular old block element so that its content doesn't get affected by the element changing size. The magic ingredient is going to be the `.content-container div` defined in Listing 7.2.

With the template defined by Listing 7.2, any page using the default layout will have a structure that looks like this (where we've added HTML comments only for clarity—they won't actually appear in the source):

```
<!DOCTYPE html>
<html>
  <head>
    .
    .
    .
  </head>
  <!-- flexbox container -->
  <body>
    <!-- 1st potential flexbox item, but not flexing because position: fixed -->
    <header>
      .
      .
      .
    </header>

    <!-- 2nd flexbox item, the only one that will be changing in size -->
    <div class="content-container">
      .
      .
      .
    </div>

    <!-- 3rd potential flexbox item, but not given a flex: property -->
    <footer>
      .
      .
      .
    </footer>
```

```
  </body>
</html>
```

To apply flexbox to our page, we'll start by adding the declaration `display: flex` to the flex container (in this case, the `body` tag). Then, in order to build the flex items in a vertical column, we'll set the *flex direction* rule `flex-direction` to `column`. Finally, we'll set the `min-height` property to `100vh` (Section cha:values_view) to ensure that the `.content-container` stretches to fill 100% of the viewport height. The resulting `body` rules (including the pre-existing `font-family` declaration) appear as follows:

```
body {
  display: flex;
  flex-direction: column;
  font-family: helvetica, arial, sans;
  min-height: 100vh;
}
```

Meanwhile, we need to arrange for the content container, which is currently shrunk down (Figure 6.21), to grow as big as it can. To do this, we need to understand the `flex-grow` property, which controls how flexbox items expand. By default, items in a flex container have a `flex-grow` value of `0`, which means the item doesn't grow at all.

Since we haven't yet applied a `flex-grow` property to any of our elements, the header, content, and footer currently don't expand to fill the container, as shown schematically in Figure 7.6.

Figure 7.6: Three flexbox items with `flex-grow` of `0`.

To get the content to fill the area, all we need to do is activate the `.content-container` as a flex item by setting `flex-grow` to `1`:

```
.content-container {
  flex-grow: 1;
}
```

The `flex-grow` property works by proportions: if all three items are set to `1`, each one takes up 1/3 of the space available. With the header and footer set to the default value of `0`, setting

the content div's `flex-grow` to `1` arranges for it to take up *all* of the available space, as shown schematically in Figure 7.7.

Figure 7.7: Arranging for only the middle item to grow using `flex-grow: 1`.

Putting together the rules for the flex container (`body`) and the flex items (header and footer set to the defaults, content with `flex-grow` set to `1`) gives the CSS shown in Listing 7.3. Note that Listing 7.3 also added padding of 10% of the viewport height (`10vh`) to move the content out from underneath the site header, which also requires a `10vh` adjustment to the height of the hero section to make it `10vh` smaller so that it still exactly fills the browser window. The width setting is there to make sure that the element stretches to fill the page horizontally as well as vertically.

Listing 7.3: *New styles for the body and a new container class.*

css/main.css

```
/* GLOBAL STYLES */
.
.
.
body {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
  font-family: helvetica, arial, sans;
}
.content-container {
  flex-grow: 1;
  padding-top: 10vh;
  width: 100%;
}
.
.
.
/* HERO STYLES */
.
.
.
.full-hero {
  background-color: #c7dbfc;
  background-size: cover;
  box-sizing: border-box;
```

```
  height: 90vh;
}
```

Et voilà ! After saving your work and refreshing the browser, the content should expand to fill the available space and push the footer to the bottom of the window. Now the padding reveals the content and fixes the problem from the end of Section 6.4, as shown in Figure 7.8.

Figure 7.8: The content now grows to fill the available space.

7.1.1 Exercises

1. Remove `position: fixed` from the header, and set both the header and the footer to `flex-grow: 1` to see how the flexbox divides space between three elements with different amounts of content.
2. Now remove `flex-direction: column` from the body to see how the browser reorders the flex items on the page.

7.2 Vertical flex centering

Our second application of flexbox is going to be styling the homepage callout section that we introduced in Section 6.3. In particular, we'll make the title and the content share the space in an aesthetically pleasing way. To accomplish this, our flexbox container will be the `div` with class `home-callout`, and the flexbox items will be the `h1` (with class `callout-title`), and the child `div` (with class `callout-copy`):

```
<div class="home-callout">
  <h1 class="callout-title">The Learn Enough Story</h1>
  <div class="callout-copy">
    .
    .
    .
  </div>
</div>
```

Our first task is to set the styles for the callout container. After initializing the flexbox with `display: flex`, we'll align the items vertically using `align-items: center`, as shown schematically in Figure 7.9.

Figure 7.9: The different ways to use the `align-items` CSS property.

The corresponding CSS appears as follows:

```
.home-callout {
  align-items: center;
  background-color: #000;
  color: #fff;
  display: flex;
  padding: 7vh 0;
}
```

Our next task is to apply the rules for the callout flex items, starting with the title. As in Section 7.1, we'll set `flex-grow` to `1` to arrange for the title to fill the available space.

We'll also set the `flex-basis`, which is a property that controls the original ("basis") width of the element. The most common values are `0`, which sets the initial width to `0` and expands only as much as is needed to contain the content, and `auto`, which automatically distributes extra space to each element. Although it's a less common usage, you can also set the flex basis to a set size value (Figure 7.10).

Figure 7.10: Flex basis can size elements proportionally, with respect to content, or to a set size.

By default, every element has a `flex-basis: auto`, but we'll change it to `0` for the title to make it fit more compactly into the container. The resulting CSS for the callout title looks like this:

```
.callout-title {
  flex-grow: 1;
  flex-basis: 0;
  font-size: 5.75vw;
  text-align: right;
  text-transform: uppercase;
}
```

Meanwhile, we'll set the properties on the `callout-copy` class to prevent the callout copy from shrinking as the parent element gets smaller by setting the `flex-shrink` property to `0` (Figure 7.11):

```
flex-shrink: 0;
```

Figure 7.11: The effect setting `flex-shrink` to `0` versus all items set to `1`.

We'll also distribute the space for the callout copy by setting the `flex-basis` to be `45em`:

```
flex-basis: 45em;
```

This leads to the following new rules for the `callout-copy` class:

```
.callout-copy {
  flex-shrink: 0;
  flex-basis: 45em;
  font-size: 0.8rem;
  .
  .
  .
}
```

Putting everything together gives the CSS in Listing 7.4. The styles in Listing 7.4 highlight the changes to existing declarations, but also notice that both the global style for the `p` and the entirely new style declaration for `.home-callout h3`. As usual, apply your technical sophistication (Box 1.2) to comment/uncomment styles to help understand what they do.

Listing 7.4: *Making the home callout look a little cooler.*

```
 css/main.css

/* GLOBAL STYLES */
.
.
.
p {
  line-height: 1.5;
  margin: 0.75em 0;
}
.
.
.
/* HOMEPAGE STYLES */
.
.
.
.home-callout {
  display: flex;
  align-items: center;
  background-color: #000;
  color: #fff;
  padding: 7vh 0;
}
.home-callout h3 {
  color: inherit;
  margin-top: 1em;
}
.callout-title {
  flex-grow: 1;
  flex-basis: 0;
  font-size: 5.75vw;
  text-align: right;
  text-transform: uppercase;
}
.callout-copy {
  flex-shrink: 0;
  flex-basis: 45em;
  padding: 0 3vw;
  font-size: 0.8rem;
}
```

When you save and refresh, you'll see that the callout section now looks a lot less haphazard and a lot more stylish (Figure 7.12).

Figure 7.12: A much better-looking callout. Flexbox all the things!

7.2.1 Exercises

1. Try out the different `align-items` values by first setting the callout flex items to `align-items: flex-start` and then `align-items: flex-end`.
2. Change the flex basis of the `.callout-copy` to `300px`. Refresh your browser to see the smaller version and how the hard value causes it not to change size.

7.3 Flexbox style options and shorthand

Now that we've seen some concrete examples, we're going to take a moment to discuss some more general aspects of flexbox. As part of this, we'll learn about a powerful shorthand notation that is the most common way to use flexbox in real-world applications. We'll use this shorthand to refactor the flexbox CSS from the previous sections, and will apply it to a three-column layout in Section 7.4.

7.3.1 Flex container properties

We'll start by illustrating the different possibilities for the flex container properties `flex-direction` and `align-items`. Figure 7.13 shows the `row` and `column` properties for `flex-direction`, as well as their `-reverse` variations. It also shows the different

possibilities for `align-items`. See if you can figure out which diagrams apply to the examples from Section 7.1 and Section 7.2 (Section 7.3.3).

Figure 7.13: Flex container properties.

7.3.2 Flex item properties

We've now seen all three of the primary flexbox item properties: `flex-grow`, `flex-shrink`, and `flex-basis`, which together allow us to control how flex items behave inside their parent container:

- `flex-grow` determines how flex items grow inside their parent. Default `0`.
- `flex-shrink` determines how flex items shrink when their parent gets smaller. Default `1`.
- `flex-basis` determines the size of flex items before space is distributed and how content is treated. Default `auto`.

The effects of these three properties are illustrated in Figure 7.14.

Figure 7.14: Flex item properties.

As illustrated in Figure 7.14, flexbox supports a shorthand notation that follows this pattern:

```
flex: <flex-grow> <flex-shrink> <flex-basis>
```

For example, as seen in Figure 7.14, the default values for the three item properties are `0`, `1`, and `auto`:

```
flex-grow: 0;
flex-shrink: 1;
flex-basis: auto;
```

In the shorthand notation, this can be written as follows:

```
flex: 0 1 auto;
```

Let's apply this shorthand to the style from Listing 7.3:

```
.content-container {
  flex-grow: 1;
  padding-top: 10vh;
  width: 100%;
}
```

Given the default values of `flex-shrink` and `flex-basis`, this is equivalent to the following:

```
.content-container {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: auto;
  padding-top: 10vh;
  width: 100%;
}
```

Thus, using the shorthand notation, we can update the CSS as shown in Listing 7.5.

Listing 7.5: *Using the flexbox shorthand for the content container item.*

```
 css/main.css

/* GLOBAL STYLES */
.
.
.
.content-container {
  flex: 1 1 auto;
  padding-top: 10vh;
  width: 100%;
}
.
.
.
```

Now let's refactor the callout CSS from Section 7.2 as well. Recall from Listing 7.4 that the callout copy class is styled as follows:

```
.callout-copy {
  flex-shrink: 0;
  flex-basis: 45em;
  .
  .
  .
}
```

Taking into account that the `flex-grow` is `0` by default, this is the same as

```
.callout-copy {
  flex-grow: 0;
  flex-shrink: 0;
  flex-basis: 45em;
  .
  .
  .
}
```

In the shorthand notation, the callout copy styles then look like this:

```
.callout-copy {
  flex: 0 0 45em;
  font-size: 0.8rem;
  padding: 0 3vw;
}
```

Finally, we can apply the same ideas to the styles for the callout title (Listing 7.4):

```
.callout-title {
  flex-grow: 1;
```

```
  flex-basis: 0;
  .
  .
  .
}
```

This is the same as

```
.callout-title {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 0;
  .
  .
  .
}
```

so the shorthand is

```
.callout-title {
  flex: 1 1 0;
  font-size: 5.75vw;
  text-align: right;
  text-transform: uppercase;
}
```

It turns out, though, that there is an even "shorterhand" way to write this, the somewhat cryptic `flex: 1`:

```
.callout-title {
  flex: 1;
  font-size: 5.75vw;
  text-align: right;
  text-transform: uppercase;
}
```

In other words, writing `flex: 1` sets both `flex-grow` and `flex-shrink` to `1`, while setting `flex-basis` to `0`.

Putting all this refactored CSS together, we end up with the compact flexbox CSS shown in Listing 7.6.

Listing 7.6: *Refactoring the home callout CSS to use short(er)hand.*

css/main.css

```
/* HOMEPAGE STYLES */
.
.
.
.callout-title {
  flex: 1;
  font-size: 5.75vw;
  text-align: right;
  text-transform: uppercase;
}
.callout-copy {
  flex: 0 0 45em;
  font-size: 0.8rem;
  padding: 0 3vw;
}
.
```

.
.

Because this was a refactoring, we changed the form of the code without changing its function, so after saving and refreshing your page's appearance should be unchanged (Figure 7.12).

7.3.3 Exercises

1. Which diagrams in Figure 7.13 apply to the examples from Section 7.1 and Section 7.2?
2. Use the `row-reverse` value for flex direction, as shown in Figure 7.13 to change the display order of the items in the `.home-callout`.
3. Now, use flex direction to change the `.home-callout` section to use the `column` value, and then `column-reverse`.

7.4 Three-column page layout

Our final application of flexbox involves creating one of the most common page layouts, something that was also one of the most difficult in the days before flexbox: creating a multi-column layout. In particular, we'll create a three-column layout where the left and right columns are a fixed width and the center column grows and shrinks as the window size changes.

You still regularly see this type of layout used for things like news stories, where you have a main content section that is flanked by navigation on the left and additional information (or subnavigation) on the right. Our three-column layout is a slight variation on this theme, which we'll end up putting to good use as a photo gallery in *Learn Enough JavaScript to Be Dangerous*.

Let's get started by adding a "gallery" link to our site navigation so that we can more easily get to the page that we created in Section 6.4, as shown in Listing 7.7.

Listing 7.7: *Adding a nav link to the gallery page.*

_includes/nav-links.html

```
<li><a href="/">Home</a></li>
<li><a href="/gallery">Gallery</a></li>
<li><a href="">Nav 2</a></li>
<li><a href="">Nav 3</a></li>
```

For the gallery layout itself, we'll use `div` s with the structure shown in Listing 7.8. This consists of a flexbox container with classes `gallery` `col-three` and three columns, each with a common `col` class and a class specialized to the purpose of each column.

Listing 7.8: *HTML for the three-column flexbox layout.*

gallery/index.html

```
---
layout: default
```

```
---

<div class="gallery col-three">
  <div class="col col-nav">
    I'm the nav
  </div>
  <div class="col col-content">
    I'm the 3col page!
  </div>
  <div class="col col-aside">
    I'm over on the right
  </div>
</div>
```

Now we'll once again use the magic of the flexbox to designate the wrapper class `.col-three` as `display: flex` to make it the flex container, and then use the shorthand notation from Section 7.3 set the properties of the columns. First, we'll set the nav column to shrink but not grow, with a minimum width (`flex-basis`) of `15em` :

```
.col-three .col-nav {
  flex: 0 1 15em;
}
```

Next, we'll arrange for the content to both shrink and grow with changing window width, with as little space as possible:

```
.col-three .col-content {
  flex: 1 1 0;
}
```

Finally, we'll have the aside column (to be used for brief image descriptions) shrink but not grow, with a flex basis of `20em` :

```
.col-three .col-aside {
  flex: 0 1 20em;
}
```

Adding in these styles together with other styles from previous sections gives the column styles shown in Listing 7.9.

Listing 7.9: *CSS for the three-column flexbox layout.*

```
 css/main.css

.
.
.
/* FOOTER STYLES */
.
.
.
/* COLUMN STYLES */
.col-three {
  display: flex;
}
.col {
  box-sizing: border-box;
  padding: 2em;
}
.col-three .col ~ .col {
  border-left: 1px solid rgba(0, 0, 0, 0.1);
}
```

```
.col-three .col-nav {
  flex: 0 1 15em;
}
.col-three .col-content {
  flex: 1;
}
.col-three .col-aside {
  flex: 0 1 20em
}
.
.
.
```

Woohoo, columns (Figure 7.15)!

Figure 7.15: Our initial three-column gallery.

Due to the rules in Listing 7.9, if you resize the browser window you'll see that the columns on the left and right stay the same size (`15em` and `20em` , respectively). Because `flex-grow` is `0` for `.col-nav` and `.col-aside` , those columns won't expand to fill the space, and since they both have a width set for the `flex-basis` , each item will be drawn as the basis width. Meanwhile, `.col-content` has `flex-grow` set to `1` and no flex basis, so it will take up as much space as it can.

All the items have their `flex-shrink` set to `1` , which means that they'll shrink in proportion as the window gets smaller, but since the basis is set for `.col-nav` and `.col-aside` they won't shrink past their basis width (Figure 7.16).

Figure 7.16: The three-column gallery in a window with a much smaller width.

But wait—our layout is currently only the height of the content... Wouldn't it be better if it were the full height of the `.content-container`? Guess how we can fix that? If you guessed more flexbox, give yourself a pat on the back!

Flexbox styles are designed to be nestable (Figure 1.14), so a flex item can also be a flex container. In this case, to get our gallery to expand in height, we can make `.content-container` —which served as a flex *item* in Section 7.1—into a flex *container* as well, via the rule `display: flex`.

If you are wondering why this won't necessarily mess up everything on other pages, remember that the default values for flex items (`flex-grow: 0`, `flex-shrink: 1`, and `flex-basis: auto`) cause them to act like regular block elements. So having the parent container initialized as a flex container isn't going to do anything weird (unless you want it to be weird... heeeyoh! (Figure 7.17)).

Figure 7.17: Teddy is down with weird flex box!

The result appears in Listing 7.10, which also applies the `flex: 1` rule used in Listing 7.6 to all three columns (`.col-three`). Recall from Section 7.3.2 that `flex: 1` is equivalent to a flex grow and shrink of `1` , so that they grow and shrink proportionally with window size, and a flex basis of `0` , so that there is a minimum of extra space.

Listing 7.10: *Arranging for the columns to take up the full height of their parent.*

```
 css/main.css

/* GLOBAL STYLES */
.
.
.
.content-container {
  display: flex;
  flex: 1 1 auto;
  padding-top: 10vh;
  width: 100%;
}
.
.
.
/* COLUMN STYLES */

.col-three {
  display: flex;
  flex: 1;
}
.
.
.
```

Saving your work and refreshing will give you an improved, full-screen, three-column layout (Figure 7.18).

Figure 7.18: Fixed gallery now takes up the whole vertical space.

7.4.1 Exercises

1. Try setting all of the columns to have `flex: 1` to make them equal in size.
2. Now try setting the left and right columns to `flex: 0 0 auto` so that they only take up as much space as the content inside requires, but also limits them from growing and shrinking.

7.5 A gallery stub

As a final step in developing our three-column layout from Section 7.4, we'll give the gallery page a quick makeover to turn our boring columns into something that looks like Figure 7.19. Adding this content and the necessary styles will both give us a chance to use some of the CSS we've been learning, and it will prep the page for the future tutorial. (In *Learn Enough JavaScript to Be Dangerous*, we'll make this page into a functional photo gallery.)

Figure 7.19: A sample of galleries to come.

This practice of filling pages with temporary material that will be made functional later is often referred to as *stubbing out content* or adding *dummy content*. In the present case, we'll need some images to stub out the gallery content, which you can get using this `curl` command:

```
$ curl -OL cdn.learnenough.com/le-css/gallery.zip
$ unzip gallery.zip -d images/        # unzip into the images directory
$ rm gallery.zip
```

This uses the `unzip` command to unzip the gallery images into the images directory (using the `-d` flag, which you can learn about using `man unzip`).[1] The result should be two new folders, `large` and `small` , each with a single image inside (Figure 7.20).

Figure 7.20: The project folders and files at this point.

Next, we're going to add the image elements into the gallery's HTML, as seen in Listing 7.11. Also, while we're adding content, we should add in some new classes on the columns so that we can more easily target gallery-related elements. Finally, we'll add in some CSS ids (Section 2.2) in anticipation of targeting them with JavaScript in *Learn Enough JavaScript to Be Dangerous*.

Listing 7.11: *Adding in dummy elements with classes and ids.*

gallery/index.html

```
---
layout: default
---

<div class="gallery col-three">
  <div class="col col-nav gallery-thumbs" id="gallery-thumbs">
    <div class="current">
      <img src="/images/small/slide1.jpg" alt="Image title 1">
    </div>
    <div>
      <img src="/images/small/slide1.jpg" alt="Image title 2">
    </div>
  </div>
  <div class="col col-content">
    <div class="gallery-photo" id="gallery-photo">
      <img src="/images/large/slide1.jpg" alt="Image title 1">
    </div>
  </div>
  <div class="col col-aside gallery-info" id="gallery-info">
    <h3>Image Title 1</h3>
    <p>Image description 1</p>
  </div>
</div>
```

If you save the file and look at the page, you'll notice that it doesn't look great (Figure 7.21).

Figure 7.21: This gallery is displaying nothing… except incompetence!

Let's add some CSS to make this page look a little better. You'll have seen most of what we are adding, but there will be a couple of new things, which we'll go over below.

On the left-hand side, you'll remember that we added a `gallery-thumbs` id and class to the `.col-nav`, and in that container we added two `div`s that both have an image in them—though one of them has the class `.current` on it. We'll set an absolute height on that column and then set the `overflow` to `scroll` so that if there are a lot of image links a user could vertically scroll through them. Then, we'll style the images in the thumbnails to be only the width of the column so that they don't take up too much space.

These thumbnails are eventually going to be the way that users switch between images in the gallery, but for this example we just copied one to show the two states that the thumbnails can be in (currently selected and default). When the `.current` class is present, the image inside has an orange-red border, and the opacity is set to `1` so that it can be clearly seen:

```
.gallery-thumbs {
  height: 90vh;
  overflow: scroll;
}
.gallery-thumbs > div {
  cursor: pointer;
}
.gallery-thumbs img {
  box-sizing: border-box;
  box-shadow: 0 0 0 5px transparent;
  display: inline-block;
  margin: 0 0 10px;
  opacity: 0.5;
  transition: all 0.5s ease-in-out;
```

```
  width: 100%;
}
.gallery-thumbs img:hover {
  opacity: 1;
}
.gallery-thumbs .current img {
  box-shadow: 0 0 0 5px #ed6e2f;
  opacity: 1;
}
```

If you hover over the other thumbnail, you'll see that it animates from partially transparent to opaque. That happens because of the `transition: all 0.5s ease-in-out` style that we added.

```
transition: all 0.5s ease-in-out;
```

That is a CSS animation style that automatically animates the way an element looks (over a period of 0.5 seconds) if its style changes. The `all` value makes it so the animation will kick in on any style (you can make it apply only to certain properties), the time is self-explanatory, and the last value `ease-in-out` tells the browser what speed it should run the animation. Ease-in-out makes it run a little slower at the beginning and end, which makes the animation feel a little more natural. (If you want to read more about the transition style, the Mozilla Developer documentation on CSS transitions has a thorough look at the property and values.)

In the center, we wrapped the big version of the image in a container with a `gallery-photo` class and id to make it easy to target. We'll use that class to set the width of the image inside to 100% so that it fills the full width of the center column:

```
.gallery-photo {
  position: relative;
}
.gallery-photo img {
  width: 100%;
}
```

On the right-hand side of the page, we added a `gallery-info` id and class to the `.col-aside` , and we'll use that class to target a couple of text related styles to make that informational section look a little nicer:

```
.gallery-info {
  font-size: 0.8rem;
}
.gallery-info h3 {
  margin-bottom: 1em;
}
```

Putting everything together gives us the gallery styles shown in Listing 7.12.

Listing 7.12: *Adding in a section of styles for our gallery.*

```
 css/main.css

.
.
.
/* BIO STYLES */
.
```

```
.
.
/* GALLERY STYLES */
.gallery-thumbs {
  height: 90vh;
  overflow: scroll;
}
.gallery-thumbs > div {
  cursor: pointer;
}
.gallery-thumbs img {
  box-sizing: border-box;
  box-shadow: 0 0 0 5px transparent;
  display: inline-block;
  margin: 0 0 10px;
  opacity: 0.5;
  transition: all 0.5s ease-in-out;
  width: 100%;
}
.gallery-thumbs img:hover {
  opacity: 1;
}
.gallery-thumbs .current img {
  box-shadow: 0 0 0 5px #ed6e2f;
  opacity: 1;
}
.gallery-photo {
  position: relative;
}
.gallery-photo img {
  width: 100%;
}
.gallery-info {
  font-size: 0.8rem;
}
.gallery-info h3 {
  margin-bottom: 1em;
}
```

When you save and refresh, you'll see that things are in order now—the gallery looks nice, changes size when you change the browser window, and reacts to your mouse cursor (Figure 7.22).

Figure 7.22: Everything in its right place.

One last little bit of cleanup before we move on. If you navigate away from the gallery and back to the homepage, you'll find that things are... not looking good (Figure 7.23).

Figure 7.23: Our flexbox in a flexbox content container has caused a layout issue.

The problem is that our `.content-container` that is now a flexbox inside a flexbox is forcing everything on the homepage to try and squish inside the wrapper horizontally. We could fix this by changing the `flex-direction` property on the `.content-container` to have a value of `column`, which would work because it would then build all the child elements vertically. Since there is no flex property set for the child elements on the homepage, they would just display as regular block elements. Unfortunately, then we'd have to lose the nice full screen gallery layout.

An easier solution, which also happens to just be a good idea in general, is to use a wrapper (yes, another! (Box 6.2)) around the whole page. In this case, we'll use a `div` and give it a class name of `home` (Listing 7.13).

Listing 7.13: *Wrapping the homepage in a div to contain all children.*

index.html

```
---
layout: default
---

<div class="home">
  <div class="full-hero hero-home">
    .
    .
    .
  </div>

  <div class="home-callout">
    .
    .
    .
  </div>

  <div class="home-section">
     .
     .
     .
  </div>
</div>
```

When you save your changes, the homepage should be back to normal (Figure 7.24).

Figure 7.24: Our homepage has returned to normal.

So why did this work, and why is it a good idea? It worked because the single wrapper that we added ends up as the only flex-eligible element inside of the `.content-container`, which means it just takes up all of the space. Inside of that, though, no flexing is being applied, so the elements just build as normal block elements and the wrapper expands naturally to contain them.

Adding a wrapper like this is a good idea because wrapping individual pages with a class that pertains to only that page can be helpful in the future when there is something that you want to style only for the content on that one page. It makes the entire homepage a portable, singular whole, instead of a collection of three child elements as it was before. We did the same thing on the gallery page when we wrapped the content in a `div` with a class of `.gallery` (Listing 7.8). Going forward, all of our pages will be wrapped with a single `div` with a class specific to that view.

7.5.1 Exercises

1. To see how the CSS transition timing property works, trying changing the `0.5s` to `2s`.

2. The transition property can also be targeted so that it pays attention only to one type of CSS change. In the `.gallery-thumbs img:hover` declaration, add in a style to set the width to 50%. When you hover over the element in the browser, the change in width will animate. Now change the transition property of the `.gallery-thumbs img` declaration to `opacity 0.5s ease-in-out`. When you save and refresh, only the opacity will animate; the width will just make a sudden change.

3. The pointer style controls what a user's cursor looks like—a helpful bit of CSS for giving users hints about what should be clickable and what should not respond. Add a new declaration targeting `.gallery-thumbs .current`, and set a style of `cursor: default`. Now when you roll over the current image the cursor will not change, suggesting to a user that the element is not clickable.

Chapter 8
Adding a blog

Now that we've polished up our site layout using flexbox (Chapter 7), the time has come to make a second Jekyll layout. This will take place within the context of adding blog capabilities to our sample website, thereby fulfilling the promise made in Section 5.3. Adding a blog to our site will give us a chance to apply many of the CSS rules we've covered so far, including fonts, margins, padding, selectors, and—you guessed it—flexbox.

Jekyll is a "blog aware" framework, which means that right out of the box it is configured to understand how to read and process content to make blog-like sites. Unlike some other blogging platforms that you might be familiar with (Figure 8.1), with a Jekyll blog there is no content management system where you go and type the content of your post into text boxes on a website. Instead, Jekyll uses individual files written in Markdown (the lightweight markup format referenced in Section 5.2 and first mentioned in *Learn Enough Text Editor to Be Dangerous*), previewed and stored in a folder on the local computer, and deployed only when ready.

The result of this design is that you can write your post using the text editor of your choice, you have complete control over the content and style of your blog, and you aren't at the mercy of a third-party platform (which could shut down at any time).[1] Moreover, by putting the project under version control Git, you also have a complete archive of the site's history. Finally, with GitHub Pages you have both an online backup and free deployment. (We use all of these tools for our own Learn Enough News site, and they are also the technologies behind the personal website of one of your authors. So you can see that static site builders are useful even when you know how to use a full-strength web development framework.)

Figure 8.1: WYSIWYG content management works great for people who are afraid of HTML and CSS.

To put together a blog with the reasonable minimum set of features, we are going to end up creating two different template views. First, we'll make a blog index page that shows some of the recent posts with a bit of preview content, and which features a sidebar that contains a list of all the recent posts (Section 8.1 and Section 8.2). We'll also adapt the flexbox methods from Section 7.4 to make a two-column layout. Second, we'll make a page for individual posts (Section 8.3), which will include the content of the blog post and a small sidebar that could be used for information about the author (or as a place to add social sharing).

8.1 Adding blog posts

We'll start by adding a sample blog post file so that our index page has something to display. By convention, Jekyll blog posts are located in a directory called `_posts`, with filenames that identify the approximate date of the post, in the format `YYYY-MM-DD-post-title.md` (year, month, date, title). (We say "approximate" because you can override the exact date in the post's frontmatter, which is especially useful for posts that start as drafts but only get published later on.) The date is used by Jekyll to automatically organize posts by the filename date, and the title of the post is used to generate the URL.

We can create the required directory and an empty post as follows:

```
$ mkdir _posts
$ touch _posts/2016-11-03-title-of-post.md
```

At this point, your site's structure should look something like Figure 8.2.

Figure 8.2: The blog index with an initial post.

The content for the stub post appears in Listing 8.1, which you should copy and paste into the post file. Note that Listing 8.1 includes extensive YAML frontmatter (Section 5.3), together with a mix of Markdown and HTML. (By design, Markdown is a *superset* of HTML, so technically every valid HTML page is also valid Markdown.)

Listing 8.1: *The first Jekyll blog post on the sample site.*

_posts/2016-11-03-title-of-post.md

```
---
layout: post
title: This is the title of the post
postHero: /images/shark.jpg
author: Me, Myself, and I
authorTwitter: http://twitter.com/mhartl
gravatar: https://gravatar.com/avatar/ffda7d145b83c4b118f982401f962ca6?s=150
postFooter: Additional information, and maybe a <a href="#">link or two</a>
---

Call me *Ishmael*. Some years ago–*never mind how long
precisely*–having little or no money in my purse, and nothing
particular to interest me on shore, I thought I would sail about a little
and see the watery part of the world. It is a way I have of driving off
the spleen and regulating the circulation.

<img class="pull-left" src="http://placekitten.com/g/400/200"
     alt="kitten">

Whenever I find myself growing grim about the mouth; whenever it is a damp,
drizzly November in my soul; whenever I find myself involuntarily pausing
before coffin warehouses, and bringing up the rear of every funeral I meet;
and especially whenever my hypos get such an upper hand of me, that it
requires a strong moral principle to prevent me from deliberately stepping
into the street, and methodically knocking people's hats off–then, I
account it high time to get to sea as soon as I can. This is my substitute
for pistol and ball.

With a philosophical flourish Cato throws himself upon
his sword; I quietly take to the ship. There is nothing surprising in this.
If they but knew it, almost all men in their degree, some time or other,
cherish very nearly the same feelings towards the ocean with me.
```

"Hold up a second," you might be saying. "What is `layout: post` doing in that file? I thought we had only the default layout. And what's all that other stuff in the frontmatter?"

It turns out that Jekyll allows layouts to be nested inside layouts, which gives us the ability to customize how you'd like all posts to be shown without needing to make a full top level layout that includes things like the HTML `head` (Listing 6.1). Then that layout can be built into the base `default` layout, which has all that necessary HTML stuff (yes, more stuff inside stuff (Figure 8.3)).[2]

Figure 8.3: Layouts in layouts, includes in includes… it's just turtles all the way down.

The rest of the content in the frontmatter are *page variables* (Box 8.1), which let us easily create and set values that can add content to the page, change how it looks, or include information (such as author name or user avatar like Gravatar) that might change from post to post. For example, in Section 8.2 we'll be able to link to the post using the title and URL defined in Listing 8.1 using the following Liquid code:

```
<h2><a href="{{ post.url }}">{{ post.title }}</a></h2>
```

This sort of per-page customization is one of many things you can't do with static HTML but which all good static site builders make easy.

Box 8.1. More on variables.

If you are thinking, "Dear Lord, more variables?! I hated algebra," don't worry. You aren't going to have to solve any equations for x. Recall from Box 6.1 that you can think of variables as boxes that you put information into—information you can retrieve by referencing the name of the box. The exact details depend on the system you're using (like Jekyll or a programming language such as Ruby), but ultimately they are just containers for information.

In the Jekyll blog post example from Listing 8.1, we are setting *page variables*, which are accessible only on the page that is loading the frontmatter. For example, consider the effect of setting a `title` variable in a page's frontmatter, like this:

```
---
title: This is the title of the post
---
```

That title can later be pulled out and placed on the page using a Liquid tag, as follows:

```
{{ page.title }}
```

This sort of flexibility allows us to define different variables for different pages, so that a single template can look different from one blog post to the next.

It is also possible to set *global variables*, which work on any page on the site. Defining global variables requires creating a new configuration file, which is beyond the scope of this tutorial, but you can read about config files and global variables in the Jekyll documentation.

8.1.1 Blog index structure

The blog itself will live in `index.html` inside a folder called `blog`, which will lead to a public URL of the form http://example.com/blog/ for the blog's index page. We can get started as follows:

```
$ mkdir blog
$ touch blog/index.html
```

For the blog index page, we'll start with a hard-coded stub version to get an idea of how it works, and will generate it automatically from the actual posts in Section 8.2. The basic design will follow the conventions we've mentioned several times before (e.g., Figure 1.2), with standard header and footers, a hero image (in this case specific to the blog), a space for the post's headline and content, and an aside box on the right-hand side with additional information (Figure 8.4).

Figure 8.4: A mockup for a blog post.

This initial stub content appears in Listing 8.2. (A couple of tags are dedented so they will fit in the listing, but you should feel free to indent it to the proper level.) Note that Listing 8.2 uses the default layout defined in Listing 6.1.

Listing 8.2: *The basic structure for our blog index page.*

blog/index.html

```
---
layout: default
---

<div class="page blog-index">
  <h1>Bloggie Blog</h1>
  <div class="col-two blog-cols">
    <div class="col col-aside blog-recent">
      <h4>Recent Posts</h4>
      <ul class="blog-title-list">
        <li>
          <a href="">Blog post title</a>
          <span>Posted: Month Day, Year</span>
        </li>
      </ul>
    </div>
    <div class="col col-content blog-previews">
```

```
      <div class="blog-posts">
        <header class="post-header">
          <h2><a href="">I am the title of an article</a></h2>
          <div class="post-byline">
<img src="https://gravatar.com/avatar/ffda7d145b83c4b118f982401f962ca6?s=150">
            <a href="#" class="social-link">Tw</a>
            by: Me, Myself, and I
            <span> - Month Day, year</span>
          </div>
        </header>
      <div class="posts-image" style="background-image:url('/images/shark.jpg')">
      </div>
        <p>
          Blurb
        </p>
      </div>
    </div>
  </div>
</div>
```

Now that we have the beginning of our blog index page, we'll need to give users a way to navigate there. Open up `_includes/nav-links.html` and change the third link to direct users to the blog (Listing 8.3).

Listing 8.3: *If a page has no link... does it exist?*

_includes/nav-links.html

```
<li><a href="/">Home</a></li>
<li><a href="/gallery">Gallery</a></li>
<li><a href="/blog">Blog</a></li>
<li><a href="">Nav 3</a></li>
```

Perfect! Now we'll be able to get to the blog from the site navigation (Figure 8.5).

Figure 8.5: We're getting good at adding new pages to the navigation!

Now let's click though to the blog's index page and see how we're doing. At this point, you probably anticipated that the page wouldn't look nice, and in fact (apart from the handsome Gravatar image) it is ugly, as expected (Figure 8.6).

Figure 8.6: Ugly page is ugly.

Let's add some styling to make the blog's index page look a little better. We'll start with a two-column layout modeled on the three-column layout defined in Listing 7.9:

```
.col-two {
  display: flex;
}
.col-two .col-aside {
  flex: 0 0 20em;
  order: 1;
}
.col-two .col-content {
  flex: 1;
}
```

Here we've defined an aside column that won't grow or shrink and has a `20em` flex basis (`flex: 0 0 20em`). The content, meanwhile, has the same `flex: 1` value used for the columns in Listing 7.10. Together, these rules arrange for one big block that will change size as the window changes, and a smaller fixed-size column on the right.

There's also an entirely new flexbox style in the `.col-two .col-aside` declaration: the `order` property. This useful CSS style allows us to arbitrarily control where on a page an element is drawn. You might have noticed in Listing 8.2 that the aside was placed *first* in the HTML—if we didn't style it to behave differently, it would show up either on the top (if we weren't using flexbox), or on the left-hand side of the page. The `order` property accepts a simple number as a value, with the default value being `0` . By setting the value to `1` , we arrange for the browser to build our aside after any elements that are in the same parent and have a lower number.

Why use `order`? We added it here because it is a powerful tool that allows us to reorder elements on a page without needing to actually move the HTML.

In addition to the flexbox rules, there are a lot of other styles that we are going to include. In the global section we will add a styling that will make all `h4` s look the same across the site:

```
h4 {
  color: #999;
  font-weight: bold;
  text-transform: uppercase;
}
```

We'll also include some `max-width` styling to the `.page` class that wraps our column layout

```
.page {
  margin: 10vh auto 15vh;
  max-width: 980px;
  width: 85vw;
}
```

These styles format the `.page` wrapper to expand to fit the window, but only until it reaches `980px` in width. Why 980px? It's an old convention that was used for site widths to make sure that they would fit into a full screen browser on a 1024x768px resolution screen (e.g., an iPad).

You can make the page bigger if your content needs the extra space, but there is something to consider if your content is mostly text: people generally find it hard to read text that is stretched too wide on a page, or text that is squished into too narrow of a wrapper. Ideally, you want to keep text to be displayed so that there are about 40 to 70 characters (including spaces) per line, and while 980px is wider than that, if you add in a side menu like we did, then there is enough space for both.

We will also style the headers on the page so that all of the content pages will have identical headers, but by constraining them to apply only inside the `.page` class we'll have the flexibility not to have to restyle anything on the homepage.

```
.page h1 {
  font-size: 3em;
  margin-bottom: 1em;
  text-align: center;
}
.page h2 {
  margin-bottom: 0.5em;
}
.page h4 {
  margin-bottom: 1em;
}
```

Finally, we have all of the blog index styling, using rules which at this point should start looking familiar. The result is a good exercise in reading CSS, and we especially recommend applying the comment-out trick mentioned in Box 1.2:

```
.blog-recent {
  text-align: right;
}
```

```
.blog-title-list {
  list-style: none;
  padding: 0;
}
.blog-title-list li ~ li {
  margin-top: 1.5em;
}
.blog-title-list span {
  color: #999;
  display: block;
  font-size: 0.8em;
  font-style: italic;
  margin-top: 0.5em;
}
.blog-posts ~ .blog-posts {
  border-top: 1px dotted rgba(0, 0, 0, 0.1);
  margin-top: 4em;
  padding-top: 4em;
}
.blog-posts .post-header {
  font-size: 0.8rem;
}
.post-header {
  margin-bottom: 1.5em;
}
.post-header img,
.post-header .social-link {
  margin-right: 0.5em;
}
.post-header img {
  border-radius: 99px;
  display: inline-block;
  height: 2.5em;
  vertical-align: middle;
}
.posts-image {
  background-position: center;
  background-size: cover;
  height: 6em;
  margin-bottom: 1.5em;
}
```

The right-hand navigation has the bullets removed from the `ul`, and a bit of text styling for the `span`s so that the information contained in them, which is just there to provide extra detail for the titles, isn't too visually distracting. We also styled the `.blog-posts` class using the general sibling selector so that multiple `.blog-posts` in a column will have some separation from each other and a faint border. Everything else was just adding margins to separate content from other content, styling the author image to be a little circle, and giving the `.posts-image` some dimensions so you can see the background image.

All together our new addition to the CSS can be seen in Listing 8.4. If you get stuck, we suggest you add piece by piece to see what they do (Box 1.2). We'll give an overview of some of the styles and their effects in a moment.

Listing 8.4: *Styling for a two-column layout using flexbox.*

css/main.css

```
/* GLOBAL STYLES */
.
.
.
h4 {
  color: #999;
```

```
  font-weight: bold;
  text-transform: uppercase;
}

/* COLUMN STYLES */
.
.
.
.col-two {
  display: flex;
}
.col-two .col-aside {
  flex: 0 0 20em;
  order: 1;
}
.col-two .col-content {
  flex: 1;
}

/* PAGE STYLES */
.page {
  margin: 10vh auto 15vh;
  max-width: 980px;
  width: 85vw;
}
.page h1 {
  font-size: 3em;
  margin-bottom: 1em;
  text-align: center;
}
.page h2 {
  margin-bottom: 0.5em;
}
.page h4 {
  margin-bottom: 1em;
}
.
.
.
/* GALLERY STYLES */
.
.
.
/* BLOG STYLES */
.blog-recent {
  text-align: right;
}
.blog-title-list {
  list-style: none;
  padding: 0;
}
.blog-title-list li ~ li {
  margin-top: 1.5em;
}
.blog-title-list span {
  color: #999;
  display: block;
  font-size: 0.8em;
  font-style: italic;
  margin-top: 0.5em;
}
.blog-posts ~ .blog-posts {
  border-top: 1px dotted rgba(0, 0, 0, 0.1);
  margin-top: 4em;
  padding-top: 4em;
}
.blog-posts .post-header {
  font-size: 0.8rem;
}
.post-header {
  margin-bottom: 1.5em;
```

```
}
.post-header img,
.post-header .social-link {
  margin-right: 0.5em;
}
.post-header img {
  border-radius: 99px;
  display: inline-block;
  height: 2.5em;
  vertical-align: middle;
}
.posts-image {
  background-position: center;
  background-size: cover;
  height: 6em;
  margin-bottom: 1.5em;
}
```

At this point, saving and refreshing should give you a blog index page with a little more polished appearance (Figure 8.7).

Figure 8.7: A more organized blog index page.

8.1.2 Exercises

1. Something cool happened in the footer of the new blog index page shown in Figure 8.6. What happened, and why? *Hint*: Recall the DRY principle (Box 1.3).

2. Many (probably most) blogs include the ability for readers to leave comments, but because Jekyll is a static site generator with no database, it isn't possible to support comments natively. Luckily, there are ways to add in comments using third-party services. Use your technical sophistication and Google-fu to see if you can find a comment system that you can drop into a static site.

3. Try setting the `order:` property for the `.col-two .col-aside` to `0`. Now change the `flex-direction` to `column`, the aside should now be at the top of the page. Now try setting it back to `1` to push it to the bottom.

8.2 Blog index content loop

Now that we have the page set up, let's fix up the index page from Listing 8.2 to generate a list of posts dynamically based on the contents of the `_posts` folder. The key is that Jekyll automatically provides a variable called `site.posts` that contains a list of posts. We can then *loop* through the posts (Figure 8.8[3] and Box 8.2) and generate the HTML corresponding to each one.

Figure 8.8: Did you say utes? No, LOOPS.

Box 8.2. What's a loop?

When you are dealing with data in a program, you often find yourself in a situation where you want to do an action a number of times until either you run out of source data, or some other condition is met.

For example, suppose you were to deal from a deck of cards until there were no cards left. In this case, dealing cards is the repeating action, and running out of cards is what stops the operation. If the card-dealing were implemented as a computer program, we might write the loop using pseudocode like this:

```
for card in deck
  deal_card(card)
end
```

The loop would terminate automatically when the deck ran out of cards.

In Jekyll, we are going to use Liquid to loop through our posts, but in later Learn Enough tutorials we'll learn how to use other languages (specifically, JavaScript and Ruby) as well. We'll see that there are slight differences in the details, but in the end they all work

basically the same way, so the material here is an excellent foundation for more general-purpose programming later on.

Our first Jekyll loop is going to look like this:

```
{% for post in site.posts %}
  ...Liquid code...
{% endfor %}
```

This is known as a *for loop* (or, in some languages, a *foreach loop*), and what it does is execute the enclosed Liquid code for each element in the site's posts.

Let's break the syntax down in more detail. The `for` loop has a starting tag that indicates the form of the loop—i.e., looping through the posts with `{% for post in site.posts %}` —and ends with the `{% endfor %}` , which causes the loop to stop when we reach the last post. At that point, Jekyll knows that we've reached the end of the loop, and will continue processing normal Markdown or HTML content.

But where does `site.posts` come from? In the introduction to this section, we described Jekyll as "blog aware", and this is part of what we mean: because of the existence of the `_posts` directory, Jekyll automatically makes a Liquid variable called `site.posts` available in our pages. Moreover, because of the date convention in the post filenames (Section 8.1), Jekyll even knows how to order them (most-recent first, as required by the conventions of blogging).

Inside of the loop, `for post in site.posts` arranges for a variable called `post` to be available for making the corresponding HTML. In this context, `post` is what's known as an *object*, which just means it gives us access to a list of standard post attributes, such as the URL and date, as well as any post attributes defined by the YAML frontmatter (e.g., Listing 8.1). The syntax to access object attributes is common to many different so-called "object-oriented" languages: simply use the name of the object followed by a dot and then the name of the attribute. For example,

```
post.url
```

is the post's URL, and

```
post.authorTwitter
```

is the author's Twitter page (a variable we added via the frontmatter in Listing 8.1).

Here's a list of the attributes we'll need in constructing the full blog post:

- `{{ post.url }}` : This looks at the post's filename and builds a URL to the post. If you click one, you'll see that the URL path is something like `http://localhost:4000/2016/11/04/second-post` . Jekyll takes the date part of the

filename and turns the year, month, and day into nested folders that can be used on the blog to do things like show all the posts from a specific day, month, or year.

- `{{ post.title }}` , `{{ post.gravatar }}` , `{{post.authorTwitter}}` , `{{ post.postHero }}` : All of these tags look at the site's front matter for variables with the same name, and then if there was information added to that variable it will put the variable's content on the page. You can add as many variables as you want.
- `{{ post.date | date: '%B %d, %Y' }}` : The date tag contains the post's date as encoded in the filename (Section 8.1) or in the frontmatter (which allows you to override the value from the filename). The content after the *pipe* character `|` tells Jekyll how we want the date to be formatted before being added to the page. There are a bunch of options for formating dates and times, and if you'd like to know more you can read about Jekyll date formatting here. In our case, the format `'%B %d, %Y'` arranges to display the date in the form `November 03, 2016` .
- `{{ post.excerpt }}` : This final tag tells Jekyll to look at the post content (everything after the frontmatter in the Markdown files) and create an excerpt by pulling out the first paragraph. If you were to leave off the `excerpt` and just add the tag `{{ post }}` then the entire post would be shown on the page.

To insert the value of any attribute, we just apply the same Liquid syntax used to insert the content in Listing 7.2. For example,

```
{{ post.url }}
```

inserts the value of the post's URL at that exact place in the Liquid template (and so on for other attributes). By comparing the stubbed-out version of the index page from Listing 8.2 with the variables listed above, see if you can figure out what the new blog index page should look like.

The HTML/Liquid result appears in Listing 8.5. Note that we've replaced the dummy post in Listing 8.2 with the loop.

Listing 8.5: *Building the blog index using a loop.*

blog/index.html

```
---
layout: default
---

<div class="page blog-index">
  <h1>Bloggie Blog</h1>
  <div class="col-two blog-cols">
    <div class="col col-aside blog-recent">
      <h4>Recent Posts</h4>
      <ul class="blog-title-list">
        <li>
          <a href="">Blog post title</a>
          <span>Posted: Month Day, Year</span>
        </li>
      </ul>
    </div>
    <div class="col col-content blog-previews">
```

```
      {% for post in site.posts %}
        <div class="blog-posts">
          <header class="post-header">
            <h2><a href="{{ post.url }}">{{ post.title }}</a></h2>
            <div class="post-byline">
              <img src="{{ post.gravatar }}" />
              <a href="{{ post.authorTwitter }}" class="social-link">Tw</a>
              by: {{ post.author }}
              <span> - {{ post.date | date: '%B %d, %Y' }}</span>
            </div>
          </header>
          <div class="posts-image"
               style="background-image:url({{ post.postHero }})"></div>
          {{ post.excerpt }}
        </div>
      {% endfor %}
    </div>
  </div>
</div>
```

Now when you refresh the page, you should see that the stubbed out content has been replaced by the information specific to the real post.

As you may already have figured out, the index page so far isn't really much of a "loop"—there's only one post, so the loop simply runs once and exits. In other words, we're not yet really using the `for` loop part in Listing 8.5.

To remedy this situation, let's copy the current post to a slightly later date:

```
$ cp _posts/2016-11-03-title-of-post.md _posts/2016-11-04-title-of-second-post.md
```

When you refresh now, you should see that there are now two posts instead of only one (Figure 8.9). This means that the `for` loop is working, and as we add new posts they will automatically appear on the blog index page.

Figure 8.9: Building the blog index dynamically by looping through posts.

Pretty cool, right? You can see how much easier it is with a site generator like Jekyll to build fairly complex sites that don't require a lot of upkeep. Instead of needing to directly edit a bunch of different pages for every blog post, you just drop a properly formatted file into the right folder and everything automagically updates.

As a final touch, let's add another loop that will add in the correct links into the menu on the right. Change `blog/index.html` file to match Listing 8.6.

Listing 8.6: *Creating a second "Recent Posts" loop.*

blog/index.html

```
---
layout: default
---

<div class="page blog-index">
  <h1>Bloggie Blog</h1>
  <div class="col-two blog-cols">
    <div class="col col-aside blog-recent">
      <h4>Recent Posts</h4>
      <ul class="blog-title-list">
        {% for post in site.posts limit:5 %}
          <li>
            <a href="{{ post.url }}">{{ post.title }}</a>
            <span>Posted: {{ post.date | date: '%B %d, %Y' }}</span>
          </li>
        {% endfor %}
      </ul>
    </div>
    <div class="col col-content blog-previews">
      {% for post in site.posts %}
        <div class="blog-posts">
          <header class="post-header">
            <h2><a href="{{ post.url }}">{{ post.title }}</a></h2>
            <div class="post-byline">
              <img src="{{ post.gravatar }}" />
              <a href="{{ post.authorTwitter }}" class="social-link">Tw</a>
              by: {{ post.author }}
              <span> - {{ post.date | date: '%B %d, %Y' }}</span>
            </div>
          </header>
          <div class="posts-image"
               style="background-image:url({{ post.postHero }})"></div>
          {{ post.excerpt }}
        </div>
      {% endfor %}
    </div>
  </div>
</div>
```

Listing 8.6 uses the same ideas as Listing 8.5; the only novelty is the use of `limit:5` to restrict the loop to show only the most recent five posts.

When you save and refresh, you should see the post names and links in the right-hand column (Figure 8.10).

Figure 8.10: Blog index... now with recent posts list!

8.2.1 Exercises

1. Using your new knowledge of Jekyll variables and how they work with loops, add in the author name and avatar to the "Recent Post" side menu.

2. Edit the second blog post that we created and change up the text that makes up the body copy, the main image, and the author name and avatar (all of which are in the frontmatter).

8.3 A blog post page

Now that we have a working blog index, we're ready to take a look at individual posts. We can get started by clicking one of the links on the blog index, which takes us to a totally unstyled page with the individual post content. As seen in Figure 8.11, the post page currently doesn't even have basic elements like the site header. Let's fix that.

Figure 8.11: The kitten is cute, but the page isn't.

The problem is that in the blog post frontmatter (Listing 8.1) we told Jekyll that these pages should be built using the `post` layout… which doesn't yet exist. The first step is to create a new file called `post.html` in the `layouts` folder, and then add in the frontmatter and Liquid tag shown in Listing 8.7.

Listing 8.7: *The most basic post page possible.*

_layouts/post.html

```
---
layout: default
---

{{ content }}
```

As promised in Section 8.1, Listing 8.7 includes a layout (`default`) within a layout (Figure 8.3).

This is the simplest that we can make the page, and it still doesn't look right, but at least the rest of the site is back, as you can see in your browser (Figure 8.12).

Figure 8.12: At least we got the rest of the site back.

What do we do next? If you guessed, "add a bunch of wrappers and styles", you just won a gold star.[4] Unlike the blog index, in Listing 8.8 we are just going to jump in and add both the HTML structure and all the Liquid tags to correctly pull information from the blog post file and build it onto the page. This is an excellent exercise in reading markup, applying both the lessons from this tutorial and from *Learn Enough HTML to Be Dangerous*.

There is one key difference between the post page and the blog index from Listing 8.5; in the case of the index, we had a loop variable called `post` (defined by us via `for post in ...`), whereas in the post itself we use a variable called `page`. This variable is supplied automatically by Liquid. The result, which you should read carefully, appears in Listing 8.8, and a brief explanation follows.

Listing 8.8: *Adding in the blog post page structure.*

_layouts/post.html

```
---
layout: default
---

<div class="post">
  <div class="half-hero" style="background-image:url({{ page.postHero }})"></div>

  <article class="page">
    <header class="post-header">
      <h1>{{ page.title }}</h2>
      <div class="post-byline">
        <img src="{{ page.gravatar }}">
        <a href="{{ page.authorTwitter }}" class="social-link">Tw</a>
        by: {{ page.author }}
        <span> - {{ page.date | date: '%B %d, %Y' }}</span>
```

```
      </div>
    </header>
    <aside class="post-aside">
      <h4>Recent Posts</h4>
      <ul class="blog-title-list">
        {% for post in site.posts limit:5 %}
          <li>
            <a href="{{ post.url }}">{{ post.title }}</a>
            <span>Posted: {{ post.date | date: '%B %d, %Y' }}</span>
          </li>
        {% endfor %}
      </ul>
    </aside>
    <div class="post-content">
      {{ content }}
    </div>
    <footer class="post-footer">
      {{ page.postFooter }}
    </footer>
  </article>
</div>
```

In Listing 8.8, we've used a new element, the `article` tag, which was created specifically to wrap sections of content that could stand on their own if you were to cut them out of the site and display them separately. So a blog post would be a good place to use `article` (as we have here), but the bios on the homepage wouldn't be (because they're specific to that page).

As shown in Listing 8.8, it's a good practice to use headers and footers inside of article to contain things like the title and byline, or (in the case of the footer) things like tags, sharing, footnotes, or anything else that would normally go at the bottom.

We've added another new HTML5 element on the page, the `aside` tag. This is supposed to be used for extra content that is related to the contents of an article. In the context of a blog, it could be a list of recent blog posts or related posts, or it could be something like links for social sharing (Twitter, Facebook, etc.).

When you save and refresh your browser, you should now see that the content has been organized and is ready for us to style (Figure 8.13).

Figure 8.13: Things are looking up.

Now let's add in a bunch of post-specific styling, as shown in Listing 8.9.

Listing 8.9: *Formatting the post page with CSS to finish the layout.*

css/main.css

```
/* BLOG STYLES*/
.
.
.
.post {
  width: 100%;
}
.post-content,
.post-footer {
  margin: auto;
  max-width: 40em;
  width: 85vw;
}
.pull-left {
  float: left;
  margin: 2em 2em 2em -2%;
}
.pull-right {
  float: right;
  margin: 2em -2% 2em 2em
}
.post-aside {
  background-color: rgba(0, 0, 0, 0.01);
  float: right;
  margin: 0 0 2em 2em;
  padding: 2em;
}
.post .post-header {
  margin-bottom: 2.5em;
  text-align: center;
}
```

```
.post-content {
  font-size: 1.1rem;
}
.post-footer {
  border-top: 1px solid rgba(0, 0, 0, 0.1);
  font-style: italic;
  font-size: 0.8em;
  margin-top: 3em;
  padding-top: 2em;
}
```

In case you are wondering why that `.half-hero` at the top wasn't showing on the post page, it isn't visible because it needs a height, and while we are editing that we will also do some background sizing and positioning, as seen in Listing 8.10.

Listing 8.10: *The* `.half-hero` *on the post page needed dimensions to be seen.*

```
 css/main.css

/* HERO STYLES */
.
.
.
.half-hero {
  background-position: center center;
  background-size: cover;
  height: 40vh;
}
```

Flipping between the two pages, you'll see that things are looking pretty organized, and the content is looking good (Figure 8.14). Notice that for the post content we bumped up the font size of the text to `1.1rem`. It is always nice to have slightly larger type for sections of content that are text-heavy for readability—a font size in the range of being equivalent to `16px` to `18px` is a good target.

Figure 8.14: Our finished blog post page.

One last little Jekyll Liquid tag trick before we move on. You might at some point want to add a link on the site, or a small description, that can direct users to the most recent blog post. Let's first add a link in the header as the last navigation link. Open up `_includes/nav-links.html` and add in the changes in Listing 8.11.

Listing 8.11: *Adding a "newest post" link to the site navigation.*

_includes/nav-links.html

```
<li><a href="/">Home</a></li>
<li><a href="/gallery">Gallery</a></li>
<li><a href="/blog">Blog</a></li>
<li><a href="{{ site.posts.first.url }}">Newest Post</a></li>
```

Now you have a link that always goes to the most recent post (Figure 8.15).

Figure 8.15: Newest post link

You can use the same formatting to pull over information from the newest post. So if you wanted to put a little preview on the homepage it would be easy. Open up the homepage `index.html`, and add in a new `.home-section`, like in Listing 8.12, right above the section with the bios.

Listing 8.12: *A preview of the newest post on the homepage.*

index.html

```
.
.
.
<div class="home-section">
  <h4>Most recent post</h4>
  <div class="blog-posts">
    <header class="post-header">
      <h2>
        <a href="{{ site.posts.first.url }}">
        {{ site.posts.first.title }}</a>
      </h2>
      <div class="post-byline">
        <img src="{{ site.posts.first.gravatar }}">
        <a href="{{ site.posts.first.authorTwitter }}"
        class="social-link">Tw</a>
        by: {{ site.posts.first.author }}
        <span> - {{ site.posts.first.date | date: '%B %d, %Y' }}</span>
      </div>
    </header>
    <div class="posts-image"
    style="background-image:url({{ site.posts.first.postHero }})"></div>
    {{ site.posts.first.excerpt }}
  </div>
</div>

<div class="home-section">
  <h2>THE FOUNDERS</h2>
  .
```

```
.
.
```

And then add in a little bit of styling to keep everything tidy (Listing 8.13).

Listing 8.13: *Styling the width of the homepage's new post preview.*

css/main.css

```
/* HOMEPAGE */
.
.
.
.home-section h2 {
  margin-bottom: 1.5rem;
  text-align: center;
}
.home-section h4 {
  margin-bottom: 0.5em;
  text-align: center;
}
.home-section .post-header {
  text-align: center;
}
.home-section .blog-posts {
  margin: auto;
  width: 75%;
}
.
.
.
```

After saving and refreshing, you now have a nicely formatted preview that will always show information from and a link to the most recent blog post (Figure 8.16).

Figure 8.16: Preview of the most recent blog post on the homepage.

8.3.1 Exercises

1. *Sacrebleu !* Is your programmer's itch acting up again? Do you see what we've repeated on both the blog index page and blog detail page? This repetition cannot stand! Refactor the site by moving the recent post list into its own include, and then load that include on both the blog index and detail pages.

2. Try playing around by adding new posts, changing the date, changing the filename to see how Jekyll builds URLs, and changing the frontmatter to have different titles, authors, etc. You now have a super-simple blog! Jekyll has a lot more bells and whistles that can make your blog even fancier, which you can learn about by reading the Jekyll documentation.

Chapter 9
Mobile media queries

At this point, our site is really coming together. We've got a nicely styled homepage, a proto-gallery with a three-column layout, and the ability to add an arbitrary number of nicely styled blog posts. In this chapter and the next, we'll add a few new levels of polish needed for a professional-grade site. We'll start adding the styles necessary to make our site look good on both desktop and mobile devices, a practice known as *responsive design*. Then in Chapter 10 we'll add miscellaneous polish like custom fonts, meta tags, and a favicon.

9.1 Getting started with mobile designs

The reason we need to add new styles for mobile devices is because websites made for big desktop screens that have lots of available space don't look great when shrunk down to be shown on small screens. Conversely, sites designed for small screens—where interactive elements need to be big for UI purposes—look terrible when blown up on a big screen (Figure 9.1).

Figure 9.1: Big on small, small on big = bad for all.

Web designers used to use a little bit of code that would detect what size screen you were using, and then serve two totally different pages depending on your screen size. Unfortunately, doing that sort of thing means that you end up needing to maintain two different sets of code… and that is the worst kind of repeating yourself (Box 1.3).

To avoid this maintenance nightmare, modern development practices use special sections of CSS called *media queries* that can be set to apply only when the screen is a certain size or has a certain attribute. An example of a media query appears in Listing 9.1.

Listing 9.1: *Applying different styles if the screen is less than 800px wide.*

css/main.css

```
.
.
.
@media (max-width: 800px) {
  html {
    box-shadow: none;
    padding: 0;
  }
  .post-aside {
    display: none;
  }
}
```

The `@media (max-width: 800px)` part of the CSS in Listing 9.1 is the media query itself, and inside of that is just regular CSS. What is happening is that if a screen is `800px` or less (called the *breakpoint*), then the browser will apply the styles that are contained inside (and recall from Table 2.1 that Media Type has a very high priority). There are a number of different media queries that let you style things so that your site looks different when printed or when seen on a mobile device held portrait vs. landscape (Figure 9.2).[1] We are going to keep things very simple, but if you want to know more, the Mozilla developer page for media queries has a lot more information.

Figure 9.2: Portrait vs. landscape.

After adding the contents of Listing 9.1 to the bottom of `main.css`, you should find that the layout changes when you change the width of your browser window (Figure 9.3). (Note that Figure 9.3 and subsequent screenshots incorporate the result of the newest post exercise (Listing 8.11).)

Figure 9.3: A bit of a difference between the bigger window and the narrower one.

The media query we added in Listing 9.1 also gets rid of the padding and black border around the site, and then also changes the `display` property of the `.post-aside` to `none` to hide it. So as you change the browser window's size, you'll see the aside disappear on the narrower screen. When you make the window bigger, it pops back into existence.

When designing content to fit on a small screen, there are often elements of a page that are nice to have if there is a ton of room but aren't strictly necessary, so we can omit them when designing for a smaller screen. In order to make good design decisions, we need to think about what is necessary to clearly communicate our content to users in a usable way, rather than trying to cram every single function of the site into the mobile version... unless of course most users will be seeing the site on mobile first (Box 9.1).

The general idea is that you should set a number of breakpoints, with different styles that get applied to elements at different window sizes, so that the entire page fits better in the window across devices.

Box 9.1. Style note: Mobile first, or desktop first?

These days, mobile traffic to sites keeps growing because smartphones are pretty much ubiquitous (sorry if you are still hanging onto that RAZR).

> Nowadays, many front-end developers actually start their styling by looking at how the page will look on mobile first, since that can be the more difficult design constraint—a practice known as *mobile-first development*.
>
> If you think that most of your users are going to be visiting your page while on a mobile device, or if you just don't know, it would be a good idea to consider starting your styling with the browser window shrunk down to approximate a mobile screen. This is especially true for content sites likes blogs or informational sites about real-world businesses—in those cases, lots of people will be visiting your site from a phone, and you should make things easy for them.
>
> However, if you have a product or service that is mostly going to be used by people who are sitting in front of a computer, it might make more sense to start with desktop design first, and then adapt down to mobile. For instance, if you are making a site that helps other developers to do better work while coding, it is pretty unlikely that they are going to be doing their coding work from a phone. In this case, you might as well start with a desktop-first design to take full advantage of the screen real estate.
>
> Like everything else in business, it's important to know your customer...

Part of mobile development is understanding how to build a site in a way that allows you to easily adapt the content for small screens with a minimum of extra styling. The nightmare scenario is one where you have to give every single element a new set of CSS styles—this would make your application incredibly brittle when it comes to handling changes without breaking the site.

We've actually done quite a bit of that in this tutorial without your noticing—there are lots of elements that are set to resize to fill a container, and we just need to sprinkle in some styling to adapt what we've built.

9.1.1 Exercises

1. Add a second break point with a new media query to target windows that are narrower than `600px`, and use it to hide the `.half-hero`.

9.1.2 How to see in mobile (without looking at your phone)

Before we get started with the full mobile restyling, we should find a way to make it easy to see our page at the approximate size of a mobile screen. We could resize the window, of course, but Chrome and Safari don't actually let you shrink the window far enough to fully approximate a mobile window. Luckily, both browsers have modes that you can turn on that exactly resize the content area of the browser, for precisely this case. Let's see how it works in Safari and then in Chrome.

In Safari, you first need to go to Preferences, and then in the "Advanced" section check the box that says "Show Develop menu in menu bar" (Figure 9.4).

Figure 9.4: Enabling the Safari developer tools.

Once the developer tools are enabled, you should see a new menu option at the top of your screen called "Develop", and in the drop-down menu there will now be an option to "Enable Responsive Design Mode" (Figure 9.5).

Figure 9.5: Enabling the responsive design mode in Safari.

When you enter that mode, your window will now show the content at a variety of mobile sizes (Figure 9.6).

Figure 9.6: The site viewed in the smaller Safari mobile view.

Now that you have the responsive design mode open, right click (or two-finger click) anywhere on the page and select "Inspect" to open up the Safari web inspector (Figure 9.7). Web inspectors

are handy tools that do a lot of things that aid developers, but most importantly for our purposes they let us see the styling that has been applied to every element on a page. Even better, they let us make test changes and see the results immediately (so that we don't always have to change the code, save the changes, and then refresh the browser).

Figure 9.7: The Safari web inspector.

Chrome has a similar mode, but you have to take a different path to enabling it. Right-click anywhere on the page to bring up the menu, and then click "Inspect". That will bring up the same sort of web inspector that we just opened in Safari (Figure 9.8).

Figure 9.8: Getting to the Chrome web inspector.

To resize the current page in a mobile view, click the little button near the top level of the inspector, the one that kind of looks like a phone in front of a page (Figure 9.9).

Figure 9.9: The sample site in the Chrome mobile view.

Now that we can see an approximation of the site on our computers, let's jump into reformatting the pages to display better on small screens.

9.2 Mobile adaptation

So, where to start with our mobile adaptation? It's a good idea to do a first round of navigating around the site using the mobile view browser setting before making any changes. Doing that give us an overview of the areas that might need attention, and usually will allow you to quickly identify parts of the site that just don't work well on mobile. Here are some of the issues to look for:

- Any element that has a set width should be given a relative width.
- Any page that has columns of content next to each other in a small space should have the content changed so that the sections stack vertically.
- Fonts should be resized for better readability.
- Details that are nice to have on the big screen can be dropped if it doesn't fundamentally affect the way users experience the site.
- Navigation that uses more exotic styling like fixed positioning should be changed to something that moves with the page so that you aren't taking up precious space. (Or, if you want the navigation always to be visible, then it is generally a good idea to make it a bit smaller for mobile.)

Let's take a look at our sample site and make a list of the areas that need to be addressed.

9.2.1 Global

- header positioning and size
- navigation layout

9.2.2 Homepage

- hero title
- bio boxes need to be vertical

9.2.3 3 Column Page

- switch to stacked layout

9.2.4 Blog Index

- drop sidebar
- have previews take up the full width

9.2.5 Blog Post

- check headline font size
- make images not float

Let's start with the header since, as seen in Figure 9.10, it doesn't quite fit.

Figure 9.10: Not a good look for the header.

In Section cha:details-dropdown and Section cha:details-mobile-dropdown, we are going to add dropdown menus to the header, and also make mobile-friendly dropdown menus that open with a click. For now, though, we'll just make everything fit. To do this, we'll add in the styles shown in Listing 9.2. Note that we've added a CSS comment to separate the blog styles from those for the header.

Listing 9.2: *Making the header work better for mobile.*

css/main.css

```
@media (max-width: 800px) {
  html {
    box-shadow: none;
    padding: 0;
  }

  /* HEADER STYLES */
  .header-nav {
    padding: 2vh 1em 0 0;
  }
  .header-nav > li {
    margin-left: 0.25em;
  }
  .header-nav > li ~ li {
    padding-left: 0.25em;
  }
  .nav-links a {
    font-size: 3.25vw;
  }
  .header-logo {
    left: auto;
  }

  /* BLOG STYLES */
  .post-aside {
    display: none;
  }
}
```

These numbers are a quick fix that we arrived at by using the code inspector in the browser to play with different values for the properties until we arrived at a decent outcome. The result is that now our site's header fits on the screen (Figure 9.11).

Figure 9.11: Reformatted header to better fit mobile screens.

Overall, that was a fairly simple edit, since all that needed to happen was a little adjustment of margins and padding. Let's now take a look at a couple of sections of the site that are going to need some reorganization in order to be readable on mobile.

Looking at the Home page, the first thing we notice is that the callout section doesn't look good any more, as seen in Figure 9.12

Figure 9.12

Scrolling down the homepage to the bio section, we'll see that the layout there also isn't well-adapted to a mobile view (Figure 9.13).

Figure 9.13: There isn't enough room for the floated bios.

The easy fix for these problems is to stack the elements on top of each other and let them take up the full width of the screen, so that both the images and the text can expand and be more readable. To accomplish this, let's add the styling shown in Listing 9.3.

Listing 9.3: *Stacking elements on mobile devices.*

css/main.css

```
@media (max-width: 800px) {
  .
  .
  .
  /* HOME STYLES */
  .home-callout {
    flex-direction: column;
  }
  .callout-copy {
    flex: 1;
  }
  .bio-box {
    float: none;
    font-size: 1.3rem;
    width: auto;
  }
  /* BLOG STYLES */
  .
  .
  .
}
```

You can see that after we got rid of the widths and dropped the `float`, all the content stretches to fill the page (Figure 9.14).

Figure 9.14: The stacked bios fit much better.

At this point, you should be getting the hang of this: You look at each page, see where something doesn't fit well, and then alter sizing so that the content better fills the screen—usually removing a width where it was set or making columns out of rows. We also hope you can see now why it is a bad idea to use tables for layouts (Box 7.1)—if we had arranged those bios using tables, there would be no way to stack them in a column.

Now let's swing through the other changes.

On the gallery page from Section 7.5, we are going to restack the elements and make the thumbnails run across the top of the image horizontally. One thing to notice: for the mobile styling, we're not going to target the column layout style classes like `.col` or `.col-three` (Listing 7.8), but rather the gallery-specific classes that we added in Listing 7.8 and Listing 7.12 (`.gallery` , `.gallery-thumbs` , etc.). The reasoning here is that there is a conceivable world where we would want to have a 3-column layout with everything horizontal even on mobile, but in the specific case of the gallery we don't want that. So instead of changing the column layout for *all* three-column layouts across the site, we're just going to target the gallery.

We'll first use `flex-direction: column` to switch the layout to be vertical, and also lessen the amount of padding on each `.col` :

```
.gallery {
  flex-direction: column;
}
.gallery .col {
  padding: 1em;
}
```

Then we'll change the `.gallery-thumbs` to get rid of the height that we set before, and instead give the element a fixed `100vw` width (Section cha:values_view) so that it will horizontally scroll if there are a lot of thumbnails:

```
.gallery .gallery-thumbs {
  flex: 1 1 0;
  height: auto;
  white-space: nowrap;
  width: 100vw;
}
```

We'll change the thumbnails to have a set height instead of a set width like they had before, update the margins and padding, and also change the flex properties for both the `.gallery-thumbs` and `.gallery-info` so that they will expand to fill space (while also no longer having a `flex-basis` that forces a set width):

```
.gallery .gallery-thumbs {
  flex: 1;
  height: auto;
  white-space: nowrap;
  width: 100vw;
}
.gallery-thumbs > div {
  display: inline-block;
}
.gallery-thumbs img {
```

```
    height: 7vh;
    margin: 0 10px 0 0;
    width: auto;
  }
  .gallery .gallery-info {
    flex: 1;
  }
```

Putting all these changes together gives the updated CSS shown in Listing 9.4.

Listing 9.4: *Horizontal gallery thumbnails with a vertical page build.*

css/main.css

```
@media (max-width: 800px) {
  .
  .
  .
  /* GALLERY STYLES */
  .gallery {
    flex-direction: column;
  }
  .gallery .col {
    padding: 1em;
  }
  .gallery .gallery-thumbs {
    flex: 1;
    height: auto;
    white-space: nowrap;
    width: 100vw;
  }
  .gallery-thumbs > div {
    display: inline-block;
  }
  .gallery-thumbs img {
    height: 7vh;
    margin: 0 10px 0 0;
    width: auto;
  }
  .gallery .gallery-info {
    flex: 1;
  }
}
```

The end result should look something like Figure 9.15.

Figure 9.15: A newly redone gallery styled for mobile.

Note from Listing 9.4 that there's a new style on the `.gallery-thumbs` that specifies `white-space: nowrap`. That style forces an element to stay on one line instead of wrapping onto two, which is important for this gallery because we want the thumbnails to be in a long horizontal row. If we hadn't added that property, and if there were a lot of thumbnails, the gallery would look like Figure 9.16.

Figure 9.16: Not the thumbnail navigation that we want, and no scrolling.

The blog needs some help too, and unlike the gallery, in this situation we are not going to rearrange the columns on the index, but rather hide the aside instead (Listing 9.5).

Listing 9.5: *Simplifying the blog index.*

css/main.css

```
@media (max-width: 800px) {
  .
  .
  .
  /* BLOG STYLES */
  .blog-recent {
    display: none;
  }
  .blog-previews {
    padding: 0;
  }
}
```

Simple removal of elements and padding took a view from looking like a mess to one that looks nice (Figure 9.17). Sometimes mobile styling ends up being pretty easy!

Figure 9.17: Just removing a couple of rules can make a big difference.

9.2.6 Exercises

1. Instead of hiding the recent posts on the blog index page, try using CSS to have them show at the very top of the pages above the post previews.

9.3 Mobile viewport

The work in Section cha:mobile-add and Section cha:mobile_adaptation was an excellent start, but in fact our site isn't quite yet mobile-ready.

There are several ways to see this. The most convenient in the long run is to install an *iOS simulator*, which lets you view a simulation of an iPhone inside of macOS, as seen in Figure 9.18. (If you're on a mac, you need to install Xcode, start Xcode from the Applications folder, then select Xcode > Open Developer Tool > Simulator.)

Figure 9.18: How it looks, and how you want it to look.

We can see the same result without using the iOS simulator by deploying the site to production, as shown in Listing 9.6.

Listing 9.6: *Deploying the mobile-ready prototype.*

```
$ git add -A
$ git commit -m "Finish mobile prototype"
$ git push
```

After waiting a moment for the site to build, using an ordinary browser (in mobile mode) to visit the production site (Listing 9.7) should show the same problem seen in Figure 9.18.

Listing 9.7: *The template for a GitHub Pages URL .*

```
https://<username>.github.io/blog
```

(There's a third, more advanced option as well, which is covered in Box 9.2.)

Box 9.2. Viewing a local server.

One alternative to using the iOS simulator (which works only on macOS) or deploying to production (which is slow and inconvenient) is to view the site directly using the local Internet Protocol (IP) address. This technique involves finding the IP address on the local network and then visiting it with your mobile device.

On many systems, the local IP can be found using the command-line program `ifconfig` ("interface configure") as follows:

```
$ ifconfig | grep 192
```

This command pipes the output of `ifconfig` to the `grep` utility covered in *Learn Enough Command Line to Be Dangerous*. The result is to pick out addresses that start with `192` , which on most networks identifies local addresses. By typing in the resulting IP number into a mobile browser and appending the proper port (Box 5.1), you can then visit the local Jekyll site.

On some systems, you'll need to quit Jekyll and restart it with a `host` argument to make the site available to other devices on the same local network, as follows:

```
$ jekyll serve --host 0.0.0.0
```

With this running, you should then be able to use something like

```
http://192.168.1.160:4000
```

to view the site locally.

Unfortunately, despite our previous efforts, it doesn't look so good—everything looks really small (Figure 9.18).[2] The reason is that, due to a quirk in the way the mobile web developed, the site loaded as though we are on a desktop, and then zoomed out to fit it all on the smaller screen.

The culprit here is a decision that was made when smartphones were first being developed. People realized that the web experience was really bad if you tried to show a standard/non-mobile-optimized site at the native resolution of the device, and because there really weren't enough phones with real browsers on the market, there also weren't many mobile-optimized sites. As a result, browser makers decided that mobile browsers could have two modes: (1) do nothing and act like a regular browser with a tiny screen size; or (2) act like a much bigger screen

and load any given website as though the user was browsing on a desktop, shrinking things down to fit as necessary.

Because phone developers didn't want to confuse the users with having to make choices about how to view a site, they created a new HTML `meta` property that lets the developer switch between modes. That way, if you've made your site mobile-optimized, you can set the viewport to show the mobile-friendly view. Otherwise, the site will load as though it were being viewed on a desktop. Since we haven't yet set that `meta` tag, the site currently appears as in the left side of Figure 9.18.

To fix the issue, all we need to do is set the tag, which controls what is called the *viewport*. The specific declaration, which should go in `_includes/head.html`, appears as shown in Listing 9.8. (There are a bunch of other options for controlling the mobile viewport that we aren't going to dive into, but rest assured you can find more info about them online.)

Listing 9.8: *The viewport `meta` tag to control the appearance on mobile.*

_includes/head.html

```
<head>
  <title>Test Page: Don't panic</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="/css/main.css">
</head>
```

This viewport setting in Listing 9.8 tells the browser that the page should be drawn at the size of the device screen instead of the virtual screen size, while setting the zoom scaling to 1 (which means no scaling). This setting will cause a page to look absolutely terrible if you try to force an unoptimized site to render at such a small size, which is why we've gone to such trouble to make our site responsive.[3]

9.3.1 Exercises

1. Try setting the `width=` to a hard pixel value like `500`, which is equivalent to 500px (the unit is left off in the viewport `meta` tag). Forcing a mobile browser to show your site only at a very specific size can be helpful when trying to develop very precise mobile layouts.

9.4 Dropdown menu

Another bit of polish we can add to show off the power of CSS is a *dropdown menu*, which are a common design pattern on the Web (Figure 9.19). In particular, when you are designing a site that has a lot of different pages, product categories, etc., you will often find yourself in a situation where you just don't have enough room in the site header to link to all the stuff that you need to

link—something especially important on tiny mobile screens. Dropdown menus are a convenient solution to this problem.

Figure 9.19: A fancy dropdown menu (with submenu) on Amazon.com.

You can theoretically get incredibly complicated with menus that use all sorts of JavaScript to do crazy things, but you can also build dropdown menus with simple techniques using only HTML and CSS. The concept is simple: you place one HTML element inside of another one, set the child to `display: none` so that it isn't visible, and then use the `:hover` pseudo-class (Section 5.7) to show the hidden element when a user hovers over the menu. Mobile is a little trickier, since there is no hover action, but we'll discuss a solution in Section cha:details-mobile-dropdown.

9.4.1 The Hitbox

To get our dropdown menu working, the first thing that we need to cover is the *hitbox*. If you play video games, you might already be familiar with the term, but if you don't, it is the area on screen where an action has an effect. In terms of a web page, this usually refers to the area of the screen where links and buttons are active—i.e., areas that will respond when touched. For example, Figure 9.20 uses the inspector tool (Section cha:mobile-view) to show the hitboxes on the homepage of our sample site.

Figure 9.20: The rectangles show where the links are active.

In Figure 9.20, you can see the rectangles around clickable elements where those elements can be activated, and those rectangles grow or shrink depending on the size of the content inside, the dimensions, and the padding. (Margins around an element don't count.)

Why is this important? Well, to make hover rollovers work you need to make sure that the hitboxes of the element that the user rolls over, as well as the dropdown menu itself, are contiguous (with no gap). Otherwise, as soon as the user tries to move the mouse to the menu, the menu will close (something that has probably spawned a million "What is wrong with my menu?" posts on the Internet). The diagram in Figure 9.21 illustrates what you want to avoid.

Figure 9.21: Mind the gap.

9.4.2 Making the dropdown

Now let's get into actually making the dropdown menu. We are first going to edit the HTML in the `_includes/nav-links` file to add in the submenu, along with a `dropdown` class on the `li`. The result appears in Listing 9.9.

Listing 9.9: *Adding in the HTML for our dropdown.*

_includes/nav-links.html

```
<li><a href="/">Home</a></li>
<li><a href="/gallery">Gallery</a></li>
<li class="dropdown">
  <a class="drop-trigger" href="/blog">Blog</a>
  <ul class="drop-menu">
    <li>Recent Posts</li>
    {% for post in site.posts limit:5 %}
      <li><a href="{{ post.url }}">{{ post.title }}</a></li>
    {% endfor %}
  </ul>
</li>
<li><a href="{{ site.posts.first.url }}">Newest Post</a></li>
```

With the HTML shown in Listing 9.9, we now have an unordered list inside the `li` of another unordered list, which is the reason why way back in Section 4.5 we targeted the styles for the navigation `li` s using the child selector; we didn't want those styles to also apply to any `li` s that are part of a nested `ul`. Also note that we left in the "Blog" link, which means that a user will still be able to click the word "Blog" and go to the blog index (or they can click the links that will show up below).

In some situations, you might want to change the link inside the `.dropdown` container to a non-link element like a `span` so that only the links in the dropdown menu are clickable. Remember in Section 2.4 when we mentioned that you might sometimes want to make a non-link element

look like a link? This is why! The reason that we're keeping the "Blog" link in this example is that in the footer we are going to block the dropdown from showing (to keep things simple down at the bottom of the site), and we'll still want the "Blog" link to be functional in that part of the page. In the interest of being complete, though, we should make this work both ways.

We'll start by updating the styling for the navigation as shown in Listing 9.10, so that the link style gets applied to `span` s or whatever else we apply the `.drop-trigger` class to.

Listing 9.10: *Making spans look like links.*

```
 css/main.css

/* HEADER STYLES */
.
.
.
.header-nav a,
.drop-trigger {
  color: #fff;
}
.nav-links a,
.drop-trigger {
  font-size: 0.8rem;
  font-weight: bold;
  text-decoration: none;
  text-transform: uppercase;
}
.
.
.
```

Let's now add CSS to make this work, including adding in a downward-pointing triangle to act as an indicator to users that this menu item triggers a dropdown menu.

What we'll do (in order of the style added) is set the `.dropdown` to `position: relative` so that the submenu can be positioned below it using `position: absolute` . Then we'll add styles to give the `.drop-trigger` a little bit of padding, and also use the `:after` pseudo-element to add a downward-pointing triangle:

```
.dropdown {
  position: relative;
}
.drop-trigger {
  display: inline-block;
  padding-right: 1.5em;
  position: relative;
}
.drop-trigger:after {
  border: 0.3em solid;
  border-color: #fff transparent transparent;
  content: "";
  height: 0;
  position: absolute;
  right: 0;
  top: 0.3em;
  width: 0;
}
```

After that, we'll style the `.drop-menu` to have a little bit of padding at the top to give it some distance from the `.drop-trigger` , and set the element to `display: none` to hide it:

```
.drop-menu {
  display: none;
  list-style: none;
  padding: 1em 0 0;
  position: absolute;
  right: 0;
  z-index: 9;
}
```

Then we'll style the `li` s and the `a` s so that they can be seen:

```
.drop-menu li {
  background-color: #fff;
}
.drop-menu a {
  color: #333;
  display: block;
}
```

Lastly, we added the style rule that displays the menu when a user hovers over an element with the `dropdown` class:

```
.dropdown:hover .drop-menu {
  display: block;
}
```

Putting everything together gives the CSS shown in Listing 9.11.

Listing 9.11: *The dropdown CSS.*

css/main.css

```
.
.
.
/* DROPDOWN STYLES */
.dropdown {
  position: relative;
}
.dropdown:hover .drop-menu {
  display: block;
}
.drop-trigger {
  display: inline-block;
  padding-right: 1.5em;
  position: relative;
}
.drop-trigger:after {
  border: 0.3em solid;
  border-color: #fff transparent transparent;
  content: "";
  height: 0;
  position: absolute;
  right: 0;
  top: 0.3em;
  width: 0;
}
.drop-menu {
  display: none;
  list-style: none;
  padding: 1em 0 0;
  position: absolute;
  right: 0;
  z-index: 9;
}
.drop-menu li {
  background-color: #fff;
```

```
}
.drop-menu a {
  color: #333;
  display: block;
}
.
.
.
```

After you save the CSS and refresh the browser, rolling over the “Blog” trigger in the header should cause the ugly but functional dropdown to appear (Figure 9.22).

Figure 9.22: Our functional dropdown menu showing on hover.

So that works... but it is really unattractive. Let’s style it up using the code in Listing 9.12 so that the menu looks presentable. A warning, this is going to be a bit complicated so that we have a nice final product! Applying the result, shown in Listing 9.12, is an excellent exercise in reading CSS.

Listing 9.12: *Improving the dropdown styling.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.
.
.
/* DROPDOWN STYLES */
.dropdown {
  position: relative;
}
.dropdown:hover .drop-menu {
  display: block;
```

```
}
.dropdown:hover .drop-trigger:after {
  border-color: #ed6e2f transparent transparent;
}
.drop-trigger {
  display: inline-block;
  padding-right: 1.5em;
  position: relative;
}
.drop-trigger:after {
  border: 0.3em solid;
  border-color: #fff transparent transparent;
  content: "";
  height: 0;
  position: absolute;
  right: 0;
  top: 0.3em;
  width: 0;
}
.drop-menu {
  box-shadow: 0 0 10px 0 rgba(0,0,0,0.2);
  display: none;
  list-style: none;
  padding: 1em 0 0;
  position: absolute;
  right: 0;
  z-index: 9;
}
.drop-menu:before {
  border: 0.6em solid;
  border-color: transparent transparent #fff;
  content: "";
  height: 0;
  position: absolute;
  right: 1em;
  top: -0.1em;
  width: 0;
}
.drop-menu li {
  background-color: #fff;
}
.drop-menu li ~ li {
  border-top: 1px dotted rgba(0,0,0,0.1)
}
.drop-menu li:first-child {
  border-radius: 5px 5px 0 0;
  color: #999;
  font-size: 0.5em;
  padding: 1em 1em 0.25em;
  text-align: right;
  text-transform: uppercase;
}
.drop-menu li:last-child {
  border-radius: 0 0 5px 5px;
}
.drop-menu a {
  color: #333;
  display: block;
  font-weight: normal;
  padding: 0.5em 2em 0.5em 1em;
  text-align: left;
  text-transform: none;
  white-space: nowrap;
}
.drop-menu a:hover {
  background-color: rgba(0,0,0,0.1);
  color: #333;
}
/* HIDE IN THE FOOTER */
.footer .dropdown:hover .drop-menu,
.footer .drop-trigger:after {
```

```
  display: none;
}
.footer .drop-trigger {
  padding-right: 0;
}
.
.
.
```

When you add Listing 9.12 into your CSS file and save the work, you'll now see a nicely styled dropdown menu when you hover over the "Blog" text in the header navigation (Figure 9.23).

Figure 9.23: Muuuuch better. Our dropdown menu no longer looks like a mistake.

From looking at Figure 9.23, it might not seem like the hitboxes of the elements are touching, due to the transparent background of the `.drop-menu` and the padding that was added. As noted in Figure 9.21, this would be a problem, but by temporarily adding in a bit of background on Figure 9.24 we can see the `.dropdown` and the `.drop-menu` are actually touching.

Figure 9.24: Our hitboxes are touching, as required.

If you want to see how it works having a non-clickable link as the target, replace the `a` tags with a span, making sure to keep the `drop-trigger` class (Listing 9.13).

Listing 9.13: *An alternate version of the dropdown where the target isn't clickable.*

_includes/nav-links.html

```
<li><a href="/">Home</a></li>
<li><a href="/gallery">Gallery</a></li>
<li class="dropdown">
  <span class="drop-trigger">Blog</span>
  <ul class="drop-menu">
    <li>Recent Posts</li>
    {% for post in site.posts limit:5 %}
      <li><a href="{{ post.url }}">{{ post.title }}</a></li>
    {% endfor %}
  </ul>
</li>
<li><a href="{{ site.posts.first.url }}">Newest Post</a></li>
```

You can use this technique for any type of hidden menu where you need to get the user to click on something in order to show the menu, like the dropdown we just made. (The trigger element can't be a link, since clicking on that would send you to page specified in the link's `href` .)

9.4.3 Exercises

1. Instead of hiding the dropdown in the footer, see if you can adapt the styling so that the menu appears above the trigger. You'll also need to flip around, and reposition, the arrows that we created using the `:after` pseudo-element.

9.5 Mobile dropdown menu

Now let's adapt the ideas from Section cha:details-dropdown to make a mobile version of the dropdown menu. The first thing that we need to do is learn about a couple of new HTML tags, `label` and `input` (in this case, with the attribute `type="checkbox"`).

We haven't used these elements before, but you've seen them practically every time that you've used the Internet. The `input` tag allows a user to input information, such as a text box, a password, or a checkbox. The `label` tag is used to give context to the `input` , e.g., `name` for inputting a name. An example appears in Figure 9.25, which shows labels and inputs on the login page for LearnEnough.com.

Figure 9.25: Labels and inputs.

The specific features that we are going to take advantage of for our mobile menu are as follows:

- The `:checked` pseudo-class (Box 9.3), which can be used to determine whether an `input type="checkbox"` has been checked or unchecked.
- The ability of a `label` to have a special `for=""` attribute. If the `for` attribute matches the CSS id of a checkbox input, then a user can set the checkbox by clicking on either the input or the text in the label.

Box 9.3. Inputs pseudo-classes.

HTML *inputs*—elements like text input fields, text areas, checkboxes, buttons, select dropdowns, etc.—all have their own pseudo-classes, which allow developers to style the different states the inputs can have as a user is interacting with them.

We aren't adding any HTML `form` tags to the sample site in this tutorial, but this table gives you an idea of some of the more common pseudo-classes and what they allow you to control:

:checked	Targets checkboxes that have been clicked by a user to add a checkmark
:disabled	Targets inputs that have been set to `disabled` so that they don't respond to user clicks
:enabled	Targets inputs that are not disabled
:focus	Targets an input that a user is actively interacting with, like when you are typing text in a text box
:invalid	Targets inputs that have entries that are invalid (to draw a user's attention

	to the element to fix their entry)
:read-only	Targets inputs that a user cannot write in, but in which they can click and select text
:valid	Targets inputs that have valid data

See the MDN discussion of pseudo-classes for more information.

To make these ideas more concrete, let's try it out on the page in Listing 9.14 *without* setting the `label`'s `for` attribute.

Listing 9.14: *Adding a label and a checkbox input.*

_includes/header.html

```
<header class="header">
  <nav>
    <input type="checkbox" id="mobile-menu" class="mobile-menu-check">
    <label class="show-mobile-menu">Menu</label>
    <ul class="header-nav nav-links">
      {% include nav-links.html %}
    </ul>
  </nav>
  <a href="/" class="header-logo">
    <img src="/images/logo.png" alt="Learn Enough">
  </a>
</header>
```

When you save and refresh your browser, you'll see that there is now the word "MENU" and a checkbox up in the top left corner. If you click on the checkbox, you'll see that it becomes checked. But if you click on the text, nothing happens (Figure 9.26).

Figure 9.26: The label and checkbox with no `for` attribute on the label.

Now in the label, add in a `for=""` attribute, and give it the CSS id name that we added to the checkbox (Listing 9.15).

Listing 9.15: *Setting the `for` attribute on the label.*

_includes/header.html

```
<header class="header">
  <nav>
    <input type="checkbox" id="mobile-menu" class="mobile-menu-check">
    <label for="mobile-menu" class="show-mobile-menu">Menu</label>
    <ul class="header-nav nav-links">
      {% include nav-links.html %}
    </ul>
  </nav>
  <a href="/" class="header-logo">
    <img src="/images/logo.png" alt="Learn Enough">
  </a>
</header>
```

Now when you click on *either* the text *or* the checkbox, the check mark gets set or unset.

Let's use our newfound ability to check and uncheck things to control elements on the page. At this point, you should switch to the mobile view in whatever browser you are using (Section cha:mobile-view).

The first thing that we are going to do is to hide the checkbox and the label for users who are not visiting the site on a mobile device. Add the code in Listing 9.16 below the dropdown section of styles in the CSS file.

Listing 9.16: *Hiding the elements that we are going to use on on mobile.*

css/main.css

```
.
.
.
/* HEADER STYLES */
.
.
.
/* MOBILE MENU */
.mobile-menu-check,
.show-mobile-menu {
  display: none;
}
.
.
.
```

Next, we are going to show these elements to users on mobile devices by adding styles in the media query that we added to the CSS file when we made changes to adapt the site to better fit small screens (Section cha:mobile-add). We are also going to hide the site header's navigation, and add a style rule to show it when the (now hidden) checkbox is checked.

The styles we added for `.show-mobile-menu` include a `display: block` to undo the `display: none` that we set for non-mobile users, and then add some padding and positioning:

```
.show-mobile-menu {
  display: block;
  float: right;
  font-weight: bold;
  margin-top: 1.5vh;
  padding: 1.5em;
  position: relative;
```

```
    text-transform: uppercase;
  }
```

We'll also add some `.header-nav` styles, which might look a little weird. Instead of hiding the navigation with `display: none`, what we'll do is make the maximum height of the element `0`, and then use `overflow: hidden` to hide the contents. Then, to show the navigation, we'll set the `max-height` of the `.header-nav` to a high number when the `.mobile-menu-check` input is checked. This lets the element expand to be as tall as the content inside:

```
.header-nav {
  max-height: 0;
  overflow: hidden;
  padding: 0;
  transition: all 0.5s ease-in-out;
}
.mobile-menu-check:checked ~ .header-nav {
  max-height: 1000px;
}
```

(We'll have more to say about the `~` rule in a moment.)

The full results appear in Listing 9.17.

Listing 9.17: *Optimizing the menu for small screens.*

`css/main.css`

```
.
.
.
@media (max-width: 800px) {
  .
  .
  .
  /* HEADER STYLES */
  .
  .
  .
  /* MOBILE MENU */
  .show-mobile-menu {
    display: block;
    float: right;
    margin-top: 1.5vh;
    padding: 1.5em;
    position: relative;
  }
  .header-nav {
    max-height: 0;
    overflow: hidden;
    padding: 0;
    transition: all 0.5s ease-in-out;
  }
  .mobile-menu-check:checked ~ .header-nav {
    max-height: 1000px;
  }
  .
  .
  .
}
```

When you refresh your browser, you now see the "MENU" label, and if you click it the site navigation will appear (though it is going to need more styling). Clicking "MENU" again will hide

the navigation (Figure 9.27).

Figure 9.27: The site navigation, hidden and visible when a user clicks "MENU".

The little bit of `max-height` styling trickery that we used, combined with the `transition` style, makes it so that the menu animates onto the page in a way that is a little more interesting than if it just appeared or disappeared.

Let's look a little closer at the functional bit of code that shows and hides the menu:

```
.mobile-menu-check:checked ~ .header-nav
```

What that is doing is applying the styles that are in the declaration to the nearby `.header-nav` (using the general sibling selector from Section 5.7.5) whenever an input with the class `.mobile-menu-check` has been checked in the same parent element. It's a neat trick that takes advantage of the fact that the sibling selector can apply styling to elements that come after the initial target element—so as long as the checkbox is above the menu in the HTML, we can add styles depending on the state of the checkbox.

So now let's make it look better. Add the styles from Listing 9.18 (including the extra class `.show-mobile-menu`) to the media query styles that we added above.

Listing 9.18: *Styling the whole mobile navigation.*

`css/main.css`

```
.
.
.
/* HEADER STYLES */
.
.
.
.nav-links a,
.drop-trigger,
.show-mobile-menu {
  font-size: 0.8rem;
  font-weight: bold;
  text-decoration: none;
  text-transform: uppercase;
}
.
.
.
@media (max-width: 800px) {
  .
  .
  .
  /* MOBILE MENU */
  .show-mobile-menu {
    display: block;
    float: right;
    margin-top: 1.5vh
    padding: 1.5em;
    position: relative;
  }
  .header-nav {
    background-color: #444;
    box-sizing: border-box;
    left: 0;
    max-height: 0;
    overflow: hidden;
    padding: 0;
    position: absolute;
    text-align: center;
    top: 10vh;
    transition: all 0.5s ease-in-out;
    width: 100vw;
    z-index: 9;
  }
  .header-nav li {
    display: block;
    margin-top: 1em;
  }
  .header-nav li ~ li {
    border: 0;
    padding: 0;
  }
  .header-nav li:last-child {
    margin-bottom: 1em;
  }
  .header-nav li:first-child a {
    color: #fff;
  }
  /* HIDE DROPDOWN IN THE NAV MENU */
  .header-nav .dropdown:hover .drop-menu,
  .header-nav .drop-trigger:after {
    display: none;
  }
  .header-nav .drop-trigger {
    padding-right: 0;
  }
  .mobile-menu-check:checked ~ .header-nav {
    max-height: 1000px;
  }
  .mobile-menu-check:checked ~ .show-mobile-menu:after {
    background-color: #000;
```

```
    color: #ed6e2f;
    content: "CLOSE";
    left: 0;
    position: absolute;
    top: 1.5em;
    width: 100%;
    text-align: center;
  }
}
```

When you save your work and refresh the browser, you should see a nice menu that animates down from the header like in Figure 9.28.

Figure 9.28: The finished site navigation.

A lot of the CSS in Listing 9.18 was just rearranging the menu so that everything builds nicely on the page, with a little dash of hiding the desktop dropdown styling (since it doesn't work on mobile) and setting the first navigational link color to white.

Also, notice that we used the same `:checked` and sibling selector trick to change the text of the button from "MENU" to "CLOSE" using an `:after` pseudo-element. It's the gift that keeps on giving!

Without this CSS trick, you would have to use JavaScript to achieve the same effect.

9.5.1 Exercises

1. Try changing the text in the `:after` psuedo-element from "CLOSE" to just an "X". Overlaying a pseudo-element with text in it is pretty much the only way to change textual content without needing to use JavaScript, and it is very handy for menus.

2. Instead of the `max-height` trick to show and hide the menu, try setting the closed height to `0` and then the open height to `90vh` to take up the entire screen. Using a big menu that fills the mobile browser can be useful for menus with lots of options.

Chapter 10
Adding more little touches

So guess what? We're pretty much done!

We've come a long way in this tutorial: we've made a well-architected site in accordance with the DRY principle (Box 1.3), added pages with things like a hero image and a three-column layout, and included some simple blogging. What else is there?

Well, if we want our site to look more professional and complete, we need to think about adding all those little details and final touches that really tie a site together, like a nice rug (Figure 10.1).[1] These include custom fonts and vector-based icons (Section cha:details-embed-links), a *favicon* (Section cha:details-favicon), and adding title and meta information to our pages so that they are better indexed by search engines (Section cha:details-description).

Figure 10.1: Nice rugs can really tie a room together.

10.1 Custom fonts

Back in Section cha:box_border-circles, we talked about changing fonts on the page using generic fonts available on virtually all systems. Now we are going to look at how to load *custom* fonts for our pages so that we aren't limited to the basic fonts our site's users happen to have installed on their computers.

You might already be familiar with what fonts are, but as a refresher: fonts are files that come with your computer (or that you can load later) that let you control how text looks. Fonts also allow you to make the text as big or as small as you like without loss of quality (as opposed to images, which look terrible if you stretch them out), and let you change text to any color your heart desires. This is because fonts aren't regular images, but rather are more like math equations (known as *vector images*). If you want them bigger, you just multiply everything by 3 (or whatever); if you want them to be a different color, you just change the color value for the part of the equation that fills in the area of the letter.

In addition to letters, fonts can also include icons, such as the classic Wingdings font (Figure 10.2) available on desktop computers via Microsoft Word.

Figure 10.2: The old-school Wingdings font.

This idea has been extended to the Web, and modern font families exist for all of the icons we need for the interactive parts of a website. These includes things like social media site logos (Figure 10.3) and buttons for the UI. Such *icon fonts* allow us to style the font items by scaling them, changing colors, etc., which is a big improvement over older, less flexible techniques using images.

Figure 10.3: Social media icons produced by icon fonts.

10.1.1 Installing vector image fonts

To use a vector image font on a website, we need to find it on the web and include it in our site. One of the most popular sources of web fonts is called Font Awesome, which has a giant list of icons that can be used for site interfaces. You can check out the catalog of icons on their site; the sample in Figure 10.4 is just a tiny fraction of the icons available.

Figure 10.4: Font Awesome has a lot of logos, interface icons, and design elements.

There are two ways that you can install fonts on a website. First, you can download the files, put them in your project, and then link to the files in the same manner that you linked the CSS file (Listing 5.7). We'll use this technique for Font Awesome. Second, you can link to the files over the Internet without actually adding them to your project. We'll use this technique in Section 10.1.2 when we install some custom text fonts.

Let's drop the Font Awesome fonts into our site and replace all the social media link text with the appropriate icons. The first step is to head to the download page on the Font Awesome site. Click the button that says download, save and extract the ZIP file (by double-clicking it), and then move it to a new `fonts` directory (created using, e.g., `mkdir fonts`). After these steps, your project directory should something like Figure 10.5.

Figure 10.5: Font Awesome files in the project folder.

Next, open up `_includes/head.html` in your text editor and include the fonts file using the same syntax as for `main.css` (Listing 10.1). Note that the version of Font Awesome is included as part of the path, so you may have to use a slightly different path if they released an update since this writing (Box 1.2).

Listing 10.1: *Linking the Font Awesome files.*

_includes/head.html

```
<head>
  <title>Test Page: Don't panic</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
        href="/fonts/font-awesome-4.7.0/css/font-awesome.css">
  <link rel="stylesheet" href="/css/main.css">
</head>
```

By the way, it's important to load custom font files *before* any other CSS links. Otherwise, the fonts won't be available when the browser tries to apply our styles.

Now let's add some icons. Open up `index.html` and replace the social links text in the hero with the markup shown in Listing 10.2. Each link contains an `i` tag with two classes, `fa` (for Font Awesome) and `fa-<logo name>`, where the logo name varies based on logo type (e.g., `fa-facebook` for Facebook's logo). Note that we've also updated the `example.com` `href`s with links to the real Learn Enough social sites.

Listing 10.2: *Replacing the old social link text with our new font icons.*

index.html

```
.
.
.
<div class="full-hero hero-home">
  <div class="hero-content">
    <h1>CODE DANGEROUSLY</h1>
    <ul class="social-list">
      <li>
        <a href="https://facebook.com/learnenough" class="social-link">
          <i class="fa fa-facebook"></i>
        </a>
      </li>
      <li>
        <a href="https://twitter.com/learnenough" class="social-link">
          <i class="fa fa-twitter"></i>
        </a>
      </li>
      <li>
        <a href="https://github.com/learnenough" class="social-link">
          <i class="fa fa-github"></i>
        </a>
      </li>
    </ul>
  </div>
</div>
.
.
.
```

Note that Listing 10.2 uses the italic tag `i` even though (as you may recall from *Learn Enough HTML to Be Dangerous*) this tag is deprecated. You can actually use `span` s to be more semantically correct if you'd like, but the people over at Font Awesome like using the old `i` tag, so we're going with them.

After saving the work in Listing 10.2, the homepage should now look like Figure 10.6.

Figure 10.6: The new icons on the page.

Although it's not important to understand the details, you may be curious to know what kinds of styles Font Awesome includes as part of `font-awesome.css`. If you open that file, you'll see style declarations like this:

```
@font-face {
  font-family: 'FontAwesome';
  src: url('../fonts/fontawesome-webfont.eot?v=4.7.0');
  src: url('../fonts/fontawesome-webfont.eot?#iefix&v=4.7.0')
  format('embedded-opentype'),
  url('../fonts/fontawesome-webfont.woff2?v=4.7.0')
  format('woff2'), url('../fonts/fontawesome-webfont.woff?v=4.7.0')
  format('woff'), url('../fonts/fontawesome-webfont.ttf?v=4.7.0')
  format('truetype'),
  url('../fonts/fontawesome-webfont.svg?v=4.7.0#fontawesomeregular')
  format('svg');
  font-weight: normal;
  font-style: normal;
}
```

Here `@font-face` is what tells the browser the name of the font (which gets referenced later in the CSS), where to find the files, what weight the font being loaded is, and the style.

Next in the Font Awesome CSS file you'll see the setup for the initial `.fa` class, which prepares an element to serve as an icon by converting the inline `i` tag into an inline block element. It also specifies that this element should use the `'FontAwesome'` font, while using a `font` shorthand that sets all properties to normal, the font size to `14px`, and the line-height to `1`:

```
.fa {
  display: inline-block;
  font: normal normal normal 14px/1 'FontAwesome';
  font-size: inherit;
  text-rendering: auto;
  -webkit-font-smoothing: antialiased;
}
```

Finally, there are a lot of more specific classes (like `.fa-twitter`) that define which icon should be shown by using the `:before` pseudo-element (Section 6.3) and using a code (`"\f099"`) that pulls up a specific icon:

```
.fa-twitter:before {
  content: "\f099"
}
```

These icons are handled by the browser just like text, so we can make them a bit bigger using plain CSS. The result appears in Listing 10.3.

Listing 10.3: *Tweaking the social links to work better with the icons.*

css/main.css

```
.
.
.
/* SOCIAL LINKS */
.
.
.
.social-link {
  background: rgba(150,150,150,0.5);
  border-radius: 99px;
```

```
  box-sizing: border-box;
  color: #fff;
  display: inline-block;
  font-family: helvetica, arial, sans;
  font-size: 1.7em;
  font-weight: bold;
  height: 1.5em;
  line-height: 1;
  padding-top: 0.25em;
  text-align: center;
  text-decoration: none;
  vertical-align: middle;
  width: 1.5em;
}
```

When you save and refresh, your icons should look like Figure 10.7.

Figure 10.7: Bigger, more easily readable icons.

If you want to play around with the size or color, or if you want to try out some of the other icons on the Font Awesome site, feel free! The name of the icon on their site's icon list is also the class name that you would need to use on the HTML element, plus an `fa-` at the beginning. For example, if you wanted to add in the "arrow in a circle pointing right" icon (Figure 10.8), the code on your page would look like this:

```
<i class="fa fa-arrow-circle-right"></i>
```

Figure 10.8: Arrow-circle-right.

10.1.2 Loading text fonts via CDN

Now that we've added an icon font from files that are linked locally, let's turn to loading a text font onto the page over the Internet from a third party's servers. There are a couple of reasons why you might want to do this. One, sometimes you aren't allowed to download fonts and store them in your project (e.g., due to copyright restrictions). Two, sometimes you want to minimize your server's file bandwidth by loading fonts via a remote link. Even Font Awesome offers the option to load their icon font via their content delivery network, or CDN (Box 10.1).

> **Box 10.1. Wherefore art thou useful, CDN?**
>
> Content delivery networks are services that solve a specific problem: if you have a file that you need users to download, and your server is the only place they can get that file, then there could be problems if your server is offline or too many people try to get the file all at once.
>
> CDNs are networks of servers that are spread across the world, and when a user requests a file the service finds the nearest server that has the needed file and sends it to the user. That helps keep the traffic on any single server low, so that all your users can get files quickly. There are CDN services that even duplicate the content of entire sites… which is exactly how GitHub Pages works. So if you've followed the standard Learn Enough sequence, you've been using advanced CDN technology ever since you first made a public page in *Learn Enough Git to Be Dangerous*!

The downside to using a CDN to load something like fonts is that you have to trust that the remote server is going to be available to all your users all the time—otherwise, the resource that you are trying to load won't be available. This is only rarely a problem, but it's worth understanding that it's one thing (of many) that can go wrong.

For our purposes, we are going to use the free Google Fonts CDN service to add a fancier *display font*, which is a design term that means it isn't intended for entire pages of text. We'll use this font, called Raleway, for headers, and then use a plainer general text font called Open Sans for the site content.

To get started, head over to the Google Fonts site and type "Raleway" in the search box. Then click the plus sign in the red circle (Figure 10.9) to select the font.

Figure 10.9: Result after searching for the Raleway font.

Next, search for "Open Sans" (not the condensed version, just the plain one) and then click the plus sign to add this font to our selection. At the bottom of the screen you'll see a black box that says "2 Families Selected"—click on that to bring up the menu that will let you configure how you'd like to embed these fonts. Then, click the red "CUSTOMIZE", and change the selected version of Raleway from the default `regular 400` to `thin 100` (Figure 10.10). What you are changing is the *weight* of the font, which is a design term for the thickness of the lines that make up the individual characters.

Figure 10.10: Customizing the fonts from Google Fonts.

Once you've switched the font weights, click "EMBED" to go back over to the view that will give you the code you need to load the font, and copy the HTML link out of the gray box, which should look like the HTML in Listing 10.4.

Listing 10.4: *The Google Fonts link HTML.*

```
<link href="https://fonts.googleapis.com/css?family=Open+Sans|Raleway:100"
rel="stylesheet">
```

This is the code needed to load the files from the Google Font CDN. It should be placed in the site's `head` tag just above the Font Awesome link, as shown in Listing 10.5.[2]

Listing 10.5: *Adding in the CDN link to the Google Fonts.*

```
_includes/head.html

<head>
  <title>Test Page: Don't panic</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://fonts.googleapis.com/css?family=Open+Sans|Raleway:100"
  rel="stylesheet">
  <link rel="stylesheet"
        href="/fonts/font-awesome-4.7.0/css/font-awesome.min.css">
  <link rel="stylesheet" href="/css/main.css">
</head>
```

That single link is everything that you need to make the custom fonts work! No dealing with files or needing to configure anything.

Now that our site is loading these font files, let's use them. In your text editor, switch back over to the `main.css` file, and change the font stack for the headers on the site to use Raleway, and the body text to use Open Sans, as shown in Listing 10.6.

Listing 10.6: *Adding a new font stack that references the custom fonts.*

```
css/main.css

.
.
.
/* GLOBAL STYLES */

body {
  display: flex;
  flex-direction: column;
  font-family: 'Open Sans', helvetica, arial, sans;
  min-height: 100vh;
}
.
.
.
h1,h2,h3,h4,h5,h6 {
  font-family: 'Raleway', helvetica, sans;
  font-weight: 100;
}
```

Notice that the names of the custom fonts in Listing 10.6 are in quotes. You don't *technically* need to wrap font names in quotes (single or double), but it's recommend by the CSS spec, and it also makes them stand out when looking through CSS styles.

Now when you save your work and refresh the page, the text in the hero should look a whole lot different (Figure 10.11).

Figure 10.11: Our much more stylish hero text.

The other headers and body copy on the page will have the new fonts too (Figure 10.12)!

Figure 10.12: Other headers and body copy with the custom fonts.

After you set up a site to use custom fonts, it is usually a good idea to click around to all the different pages and see if anything looks weird. Different fonts can have very different default sizes, the weights of fonts might not look good at different sizes, and the custom fonts might use a completely different line height than what was being used before. Your fonts can easily look messy and in need of tweaking.

So that's custom fonts! They're easy to set up, and there are a ton of different font services out there, both free and paid. No matter what the service is, you now know how to use custom fonts, whether they're downloaded into your project and served locally or loaded over the Internet via CDN.

10.1.3 Exercises

1. Now that you've seen how to add icons, make sure that all of the social links across your site are using an icon. (Unlike most other exercises, you should keep these changes.)
2. Try loading up another font from Google Fonts, and then set the `.bio-box h3` s to use the new font.

10.2 Favicons

One little detail that we can't resist including in this tutorial is showing how to add the *favicon* ("fave icon"), a small image identifying the site that shows up on tabs or in the favorites list (Figure 10.13). As with font declarations, favicons get included in the `head` section of the page.

Figure 10.13: A page's favicon.

Favicons used to be a real pain to make due to Microsoft's old browsers only accepting a funky filetype called `.ico` , but nowadays all we need is a simple `.png` (for Portable Network Graphics, pronounced "ping").[3]

Favicon images should be square, with a side-length that is an integer multiple of 8. (The size on most browsers is 16px x 16px, so making sure it cleanly can be resized down is important, but don't go bigger than 144px or so). To see how it works on our site, we've prepared a Learn Enough–themed favicon, which you can download as follows:

```
$ curl -OL cdn.learnenough.com/le-css/favicon.png
```

(If you want to use a custom favicon, just replace the result of the `curl` command with an image with valid dimensions with the name `favicon.png` in the root directory of the project.)

Then include it in the `head` using the `link` tag, as shown in Listing 10.7.

Listing 10.7: *Adding the link for the favicon.*

_includes/head.html

```
<head>
  <title>Test Page: Don't panic</title>
  <link href="/favicon.png" rel="icon">
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://fonts.googleapis.com/css?family=Open+Sans|Raleway:100"
  rel="stylesheet">
  <link rel="stylesheet"
        href="/fonts/font-awesome-4.7.0/css/font-awesome.min.css">
  <link rel="stylesheet" href="/css/main.css">
</head>
```

After you save your work, you might be disappointed to find that when you refresh your browser you can't see the icon, but don't worry—it's there. The inability to notice a new favicon is an annoyance that still hasn't been fixed in modern browsers—unless you completely clear the browser cache, they don't refresh the favicon of visited sites for a long time. You can clear the cache, of course, but it's often inconvenient to lose all that history, and we suggest using a browser (e.g., Firefox or Brave) that you haven't yet used to visit the sample site. Either way, the favicon should appear as in Figure 10.14.

Figure 10.14: Trust us, it works...

There are other types of icons that you can add, one of which is the "Apple touch icon". This icon is what you see if you save a web page as a favorite in Safari, or to the menu of apps on iOS. (It also works on Android.) To make one, you'll need a square image that is at least 180px x 180px and saved as `apple-touch-icon.png` in the root of your site. Then link it in the `head` of your site like this:

```
<link rel="apple-touch-icon" href="/apple-touch-icon.png">
```

Adding an Apple touch icon to the sample site is left as an exercise (Section 10.2.1).

10.2.1 Exercises

1. Find a square image on the Internet, save it as `apple-touch-icon.png` in your root folder, and then link the image file.

10.3 Custom title and meta description

As a final touch, we'll add two per-page customizations that most users won't notice, but which are useful for any computer programs that interact with our site—especially the *web spiders* used by search engines to crawl the Web (Figure 10.15). The resulting custom `title` tag and `meta` description are especially important for Search Engine Optimization, or SEO (Box 10.2).

Figure 10.15: Page title and description in search results.

Box 10.2. SEO in one box.

Although sometimes considered to be a complicated subject, the basics of Search Engine Optimization, or *SEO*, are quite simple—(almost) simple enough to fit in a single tweet:

Because of Twitter's (at the time) 140-character limit, I couldn't *literally* fit all of SEO into a single tweet, but it's only a slight exaggeration, and you can fit nearly all of it into one aside box. In particular, in line with the Pareto Principle (also known as the 80/20 Rule), you can get most of the benefits of SEO with a tiny fraction of the effort.

Step 2 from the tweet—making something people want to link to—is up to you: undeniably a challenge, and beyond the scope of this tutorial. But Step 1 is easy: just make sure to include most or all of the words you want to target for search results in the `title` tag (Section 10.3.1), `meta` tags (Section 10.3.2), and headings (especially `h1` and `h2`).

For example, suppose you wanted to target the terms “pi is wrong”, “tau day”, “the tau manifesto”, and “michael hartl”. Then a good title might look like this:

```
<title>Tau Day | No, really, pi is wrong:
The Tau Manifesto by Michael Hartl</title>
```

An `h1` with “No, really, pi is wrong: The Tau Manifesto” and a `meta` description starting with “The Tau Manifesto is dedicated to one of the most important numbers in mathematics…” would complete the basic SEO requirements.

If you have the luxury of tailoring your domain name to the keywords you’re targeting, an exact domain match can also help a lot. This step isn’t always possible, but when it is it can work exceptionally well, as it did with, e.g., the Rails Tutorial. And even if you don’t have an exact domain match, you can still get a lot of value out of “pretty URLs” (Section 6.4) that include the keywords you’re targeting, such as learnenough.com/css-and-layout-tutorial.

Note that everything above, though tailored a bit to search engines, is completely honest: there’s nothing here that’s not related to our site’s mission, and keywords aren’t repeated to excess. This philosophy applies even more to the main content of the page—if you’re actually writing about the subject whose keywords you’re targeting, the right word usage will emerge organically, so don’t worry about the details of your page copy. Indeed, search engines typically punish attempts to artificially manipulate the results, so any such efforts are likely to backfire.

Finally, when it comes to getting inbound links, quality content is king. Attempts to game search engines (with reciprocal link exchanges, link farms, etc.) are punished even more severely than attempts to manipulate the page copy, so don’t even try it.

SEO isn’t exactly rocket science. Using only the techniques in this box, we’ve made sites with highly ranked search results for the English words “softcover” and “coveralls”, the phrase “learn enough”, the competitive search term “rails tutorial”, the phrase “pi is wrong”, and even a letter in the Greek alphabet. Good luck!

10.3.1 Custom title

So far in this tutorial we’ve left the page title alone and kept it the same for all pages, but Jekyll gives us a way to customize it on a per-page basis. This is useful for things like blog posts, whose subjects might well be the target of searches on keywords included in the title (Box 10.2).

The idea is to include a custom page title if there is one (along with the string `| Test Page`, indicating it’s still a page on the test site). If there isn’t such a custom title, we’ll simply use the original default title defined in Listing 1.4 (`Test Page: Don't Panic`).

The way we’re going to do this is with some Liquid code that uses a *conditional*, which executes one or another branch of code, depending on whether a given condition is satisfied. It looks like this:

```
{% if page.title %}
  <title>{{ page.title }} | Test Page</title>
{% else %}
```

```
  <title>Test Page: Don't Panic</title>
{% endif %}
```

This code looks for the presence of the `page.title` variable first seen in Listing 8.8. If it exists, the condition is `true`, and the first branch will be executed, which inserts the page title followed by `| Test Page`:

```
<title>{{ page.title }} | Test Page</title>
```

For example, Listing 8.1 sets the title of a blog post to "This is the title of the post", so the inserted title would look like this:

```
<title>This is the title of the post | Test Page</title>
```

If `page.title` hasn't been set, the condition is `false`, and the second branch will be executed, thereby inserting the default title:

```
<title>Test Page: Don't Panic</title>
```

Putting the conditional into the `head` section of the site gives the code shown in Listing 10.8.

Listing 10.8: *Including a custom title (if present).*

_includes/head.html

```
<head>
  {% if page.title %}
    <title>{{ page.title }} | Test Page</title>
  {% else %}
    <title>Test Page: Don't Panic</title>
  {% endif %}
  <link href="/favicon.png" rel="icon">
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://fonts.googleapis.com/css?family=Open+Sans|Raleway:100"
  rel="stylesheet">
  <link rel="stylesheet"
        href="/fonts/font-awesome-4.7.0/css/font-awesome.min.css">
  <link rel="stylesheet" href="/css/main.css">
</head>
```

Once you save Listing 10.8, you should see that the homepage title stays the same old "Test Page: Don't Panic", but if you go to the most recent blog post, the title of the blog post will be the first part of the title (Figure 10.16). As noted above, this is because of the custom `title` defined in the frontmatter of Listing 8.1.

Figure 10.16: A page without a set title, and one with.

On any page where we want a custom title, we can now easily add one just by adding a line to the frontmatter. For example, to use "Dangerous Blog" as the name of our blog, we can add a line to the frontmatter of the blog's index page, as shown in Listing 10.9.

Listing 10.9: *Adding a variable to the frontmatter of a page to change the title.*

blog/index.html

```
---
layout: default
title: Dangerous Blog
---
```

Save the work, refresh the browser, and success (Figure 10.17)!

Figure 10.17: Adding a custom page title.

10.3.2 Custom descriptions

Now we'll use the same conditional technique to include a custom description in our site's header. We'll use the `meta` tag, which so far we've used for defining the character set (Listing 1.4) and the viewport (Listing 9.8). For this third application, the bit of meta-information added will be a textual description of our site. This change will not be visible to regular users, but it will appear in the page's source, and as noted above is used by automatic programs like web spiders.

We'll use the same basic technique used for the `title` tag: if a page description is present, we'll include a `meta` tag with content equal to that description, otherwise (`else`) we'll include a default description. The result appears in Listing 10.10.

Listing 10.10: *Adding a custom description.*

_includes/head.html

```
<head>
  {% if page.title %}
    <title>{{ page.title }} | Test Page</title>
  {% else %}
    <title>Test Page: Don't Panic</title>
  {% endif %}
  {% if page.description %}
    <meta name="description" content="{{ page.description }}">
  {% else %}
    <meta name="description" content="This is a dangerous site.">
  {% endif %}
  <link href="/favicon.png" rel="icon">
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://fonts.googleapis.com/css?family=Open+Sans|Raleway:100"
  rel="stylesheet">
  <link rel="stylesheet"
        href="/fonts/font-awesome-4.7.0/css/font-awesome.min.css">
```

```
  <link rel="stylesheet" href="/css/main.css">
</head>
```

At this point, we can add a description variable to any page's frontmatter and the description will be loaded for that page only. For example, to make the blog's description more expressive, all we need is an extra line in the frontmatter, as shown in Listing 10.11.

Listing 10.11: *Adding a variable to the frontmatter of a page to change the title.*

blog/index.html

```
---
layout: default
title: Dangerous Blog
description: A dangerous site deserves a dangerous blog.
---
```

Because the `meta` tag isn't visible on the site, you won't notice any change even after saving and refresing, but if you view the page's source code you'll see the custom description in the `head` section of the site (Figure 10.18).

Figure 10.18: Adding a custom page-specific meta description.

Changing the title and the `meta` description are just a couple of the informational elements that you can set for a site, but since they show up in search results they are definitely two of the most common ones. The Open Graph protocol includes a more complete list of common `meta` elements.

In general, all of these meta information tags work pretty much the same way, and you'll want to set up a default version for any page that doesn't need to have anything custom, using frontmatter variables to define the content on specific pages. Just make sure that the variable names used in the Liquid tags (e.g., `page.description`) match the ones in the frontmatter (e.g., `description:`) and you're golden—you'll have learned how to fish rather than having a fish given to you (Figure 10.19).[4]

Figure 10.19: Learning how to fish.

10.3.3 Exercises

1. Add in title and description Open Graph meta tags onto your site, and have them use the same title and description variables as the old-school meta tags that we added in this section.

10.4 Conclusion and further reading

Congratulations! You have now learned enough CSS & Layout to be *dangerous*.

Now that the sample site is finished, it's time for one final deployment:

```
$ git add -A
$ git commit -m "Finish the sample site"
$ git push
```

Let's take a moment reflect on just how far we've come. We started this tutorial with the bare-bones site shown in Figure 10.20. By writing good semantic markup (Chapter 2), using CSS values (Chapter 3), applying the box model (Chapter 4), and using a static site builder for proper DRY layout (Chapter 5 and Chapter 6) with flexbox (Chapter 7), we transformed that plain page into the site shown in Figure 10.21.

Add to that the beginnings of a blog (Chapter 8), a mobile-friendly view (Chapter 9), and the little touches from this chapter, and we've got a great foundation for a personal site or a company website, featuring a blog and as many custom pages as we care to make.

Quite a difference!

Figure 10.20: Before Learn Enough CSS & Layout to Be Dangerous.

Figure 10.21: After Learn Enough CSS & Layout to Be Dangerous. Muuuuch better!

As much as we've learned in this tutorial, there's still more to do. As noted in Section 1.1, *Learn Enough CSS & Layout to Be Dangerous* is a front-end development tutorial, not a design tutorial

per se, but it's a tremendous head-start for learning more about web design.

On the subject of design, it's hard to give specific recommendations, because everyone we know cobbles together their knowledge from here and there. Our best advice is to work on a project you care about, Google around (Box 1.2) to figure out how to do it, and keep practicing. That's the only way to get better.

On the development side, there's lots more to learn about Jekyll, including aspects of the system that are more closely related to full-strength web development (such as so-called "Gemfiles", command-line commands like `jekyll new`, etc.). The Jekyll documentation is a good place to start, and the Learn Enough Ruby sequence (starting with *Learn Enough Ruby to Be Dangerous*) will also help you level up.

Although the site we've created in this tutorial is fully functional, there are a few details needed to make the sample site 100% professional grade, mostly related to using a custom domain like www.example.com instead of <username>.github.io. This involves buying a custom domain, configuring its DNS settings, setting up a custom email service, and so on. Once you do these things, your site will live at a URL like www.example.com, and your email address will be something like your.name@example.com. These details and more are covered in *Learn Enough Custom Domains to Be Dangerous*.

Finally, let's talk about next steps. With the material in this tutorial, you now have the HTML and CSS knowledge (especially CSS classes and ids) to start learning *JavaScript*, the native programming language of web browsers. This is the subject of the next tutorial in the Learn Enough introductory sequence, *Learn Enough JavaScript to Be Dangerous*.

After that, you'll be ready to start learning web development itself, via the Intro Ruby Web Development trilogy (*Learn Enough Ruby to Be Dangerous*, *Learn Enough Sinatra to Be Dangerous*, and *Learn Enough Rails to Be Dangerous*), which in turn prepares you for the full *Ruby on Rails Tutorial*.

1. **Developer Fundamentals**
    1. *Learn Enough Command Line to Be Dangerous*
    2. *Learn Enough Text Editor to Be Dangerous*
    3. *Learn Enough Git to Be Dangerous*
2. **Web Basics**
    1. *Learn Enough HTML to Be Dangerous*
    2. *Learn Enough CSS & Layout to Be Dangerous* (you are here)
3. **Beginning Development**
    1. *Learn Enough JavaScript to Be Dangerous*
    2. *Learn Enough Ruby to Be Dangerous*

4. **Professional Web Development**

- *The Ruby on Rails Tutorial*

To get the most out of these and other Learn Enough tutorials, we suggest joining the Learn Enough Society, a subscription service with full access to the online tutorials, streaming videos, and a private Slack chat group. (In addition, if cost is an issue, scholarships are also available.)

We hope you've enjoyed *Learn Enough CSS & Layout to Be Dangerous*. Now get out there and start making things!

---

1. Image retrieved from https://www.flickr.com/photos/nadiapriestley/495707437 on 2017-05-22. Copyright © 2007 by Nadia and used under the terms of the Creative Commons Attribution-ShareAlike 2.0 Generic license.
2. The pipe character `|` in the HTML supplied by Google Fonts actually breaks HTML validation. This shouldn't ever matter, as all modern browsers will interpret the code correctly, but if you care you can replace `Open+Sans|Raleway` with `Open+Sans%7CRaleway` in Listing 10.5, where `%7C` is the URL encoding for `|`.
3. In classic hacker style, the original full name for PNG was actually PING, a recursive acronym for "PING is not GIF".
4. Image retrieved from https://www.sketchport.com/drawing/6000777126477824/teach-a-boy-to-fish on 2017-05-31. Copyright © 2014 by Kanani and used unaltered under the terms of the Creative Commons Attribution 4.0 International license.

LEARN ENOUGH

THE LE STORY NEWS COURSES LOG IN REGISTER

ABOUT LEGAL CONTACT